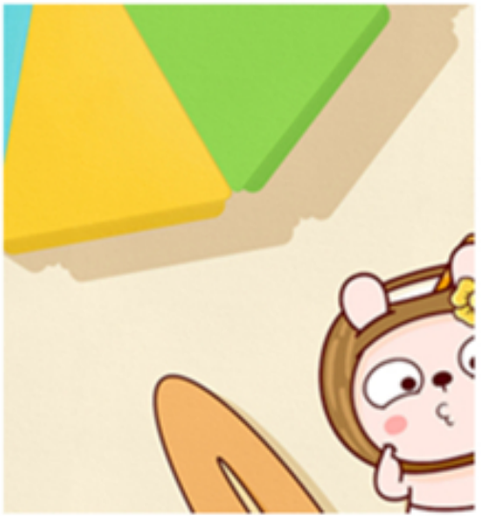
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

累计销量 10万册

JWorld@TW技术论坛版主
Java权威技术顾问与专业讲师

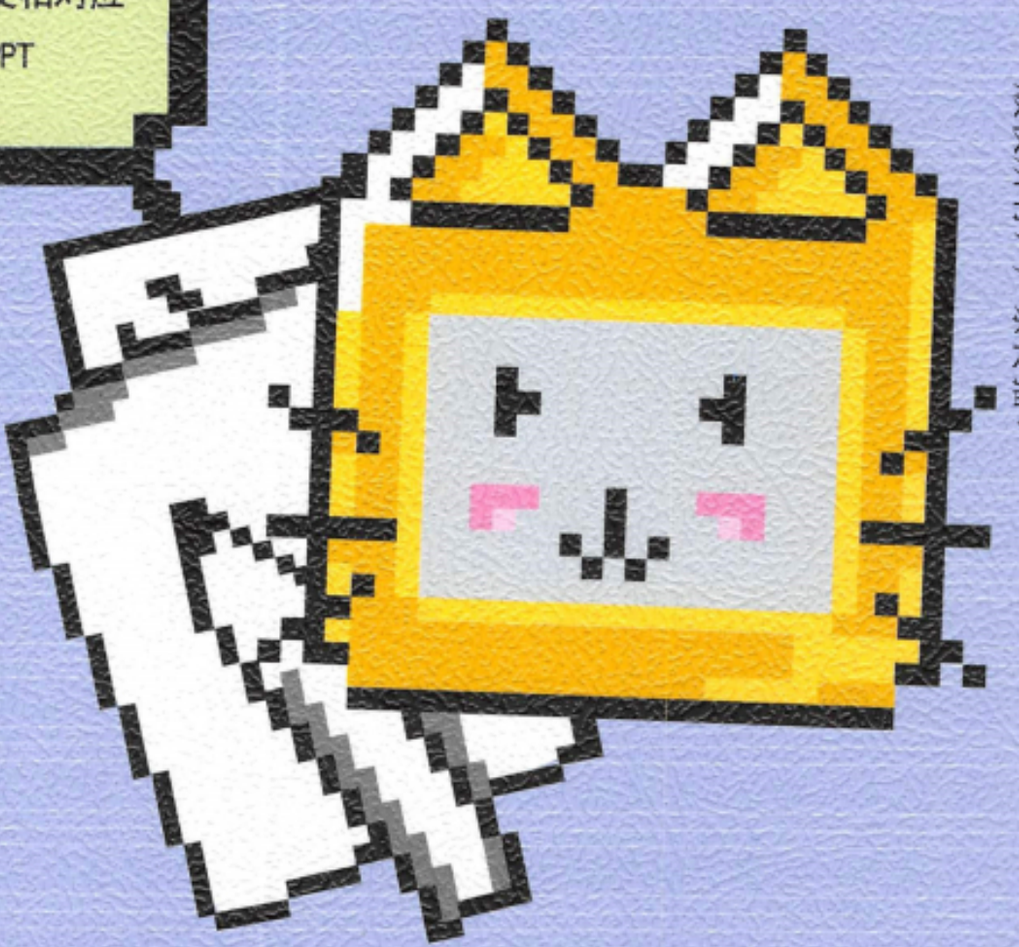
全新改版！

Java 学习笔记

JDK 9

- 分享作者学习Java心得
- 涵盖OCP/JP(原SCJP)考试范围
- 提供Java 9新功能快速索引
- 深入探讨Java模块平台系统
- JDK基础与IDE操作交相对应
- 提供Lab文档与教学PPT

林信良 编著



非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF！
版权所有，严禁传播！

碁峰 www.gotop.com.tw
严禁网络传播本PDF，违者责任自负！
清华大学出版社
非卖品，仅供非商业用途或交流学习使用
版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

林信良(网名：良葛格)，毕业于台湾大学电机工程学系。擅长技术写作、翻译与教育培训。喜好研究程序语言、框架、社群，从中学习设计、典范及文化，并利用闲暇之余记录所学技术，涵盖C/C++、Java、Ruby/Rails、Python、JavaScript、Haskell、OpenSCAD等领域。目前出版的著作有《JSP & Servlet学习笔记(第2版)》《Spring技术手册》《Java JDK 8 学习笔记》等，译作有《Ajax实战手册》《jQuery实战手册(第2版)》等。

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF！
版权所有，严禁传播！

严禁网络传播本PDF，违者责任自负！

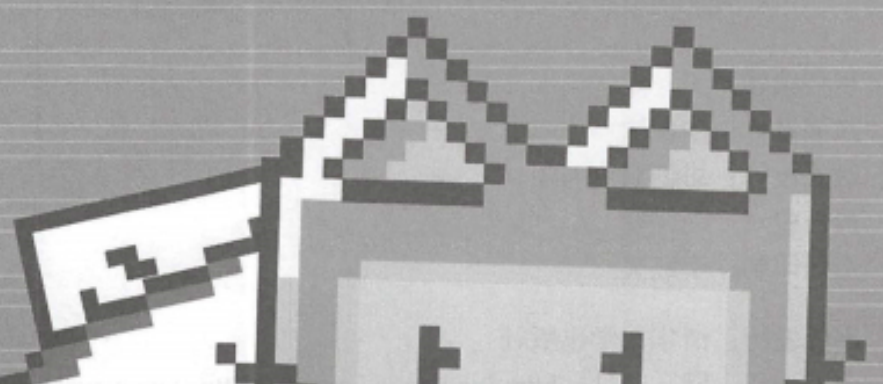
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

JDK 9

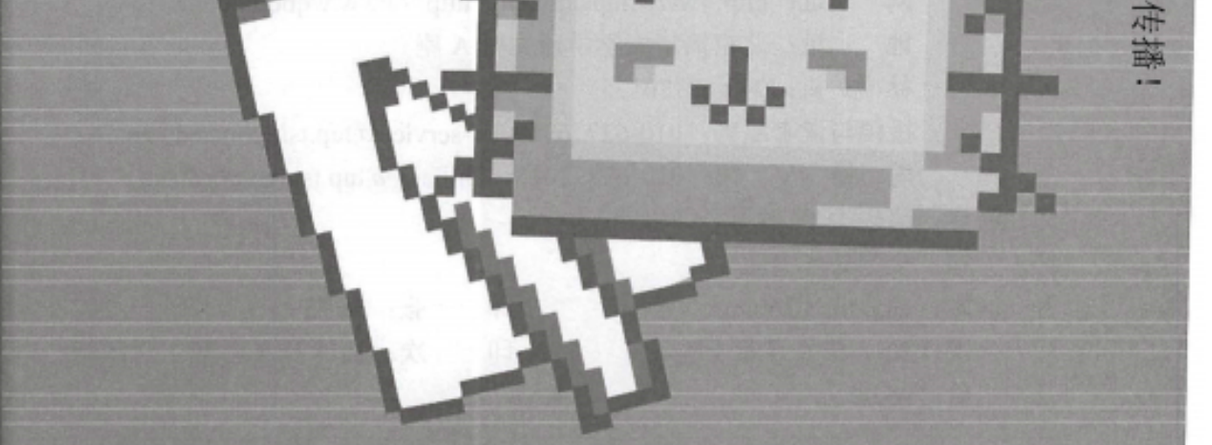
Java 学习笔记

林信良 编著



非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF！ 版权所有，严禁



清华大学出版社

北 京

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

内 容 简 介

本书是作者多年来教学实践经验的总结，汇集了学员在学习 Java 或认证考试时遇到的概念、操作、应用等问题及解决方案。

本书针对 Java SE 9 新功能全面改版，无论是章节架构或范例程序代码，都做了重新编写与全面翻新，并详细介绍了 Java 9 的模块化，JVM、JRE、Java SE API、JDK 与 IDE 之间的对照关系。必要时可从 Java SE API 的源代码分析，了解各种语法在 Java SE API 中如何应用。对于建议练习的范例提供了 Lab 文档，以突出练习重点。此外，本书还将 IDE 操作纳为教学内容之一，让读者能与实践相结合，轻松快速掌握 Java 编程技巧。

本书适合 Java 的初、中级读者以及广大 Java 应用开发人员阅读。

本书资料可通过 http://www.tupwk.com.cn/downpage 免费下载。

北京市版权局著作权合同登记号　图字：01-2018-3785

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

Java JDK 9 学习笔记 / 林信良编著. —北京：清华大学出版社，2018
ISBN 978-7-302-50118-3

Ⅰ. ①J…　Ⅱ. ①林…　Ⅲ. ①JAVA 语言－程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2018)第 100148 号

责任编辑：王　定
封面设计：牛艳敏
版式设计：思创景点
责任校对：孔祥峰
责任印制：李红英

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF！　版权所有，

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　印　　张：36.75　　字　　数：941 千字
版　　次：2018 年 6 月第 1 版　　印　　次：2018 年 6 月第 1 次印刷
定　　价：98.00 元

产品编号：079536-01

严禁传播！
严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用
版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

序

当你拿起这本书，翻开这篇序，我便有了机会问你一个问题：“为什么想翻开这本书？”

“这个梗上一版的序用过啦！”

好吧！那学 Java 目的是什么呢？从事程序设计？那么我就有个新梗了：“什么是‘设计’呢？”

唔！好难回答的问题，来拜一下 Google 大神吧！搜索“什么是设计”，看完答案之后就更不懂了，在没有一个具体目标之前，得到的可能都是近乎空洞的答案吧！

试着在身边找个东西，比如键盘，你觉得设计得好吗？不好？哪里觉得不好？触感！什么样的是不好？反馈的力道！反馈是来自哪里？键轴！键轴是由哪些成分组成的呢？底座、弹簧、轴心、轴帽！造成反馈差异性的主要来源是什么？弹簧和轴心！喔？弹簧啊？它的圈数是多少

非卖品，
严禁

仅供非商业用途或交流学习使用

呢？……

当你逐一挖掘出其中的元素之后，面对一个觉得设计不错的键盘，或许你就能知道其中有哪些“设计”了。

很多时候，当谈到一件东西设计得好或不好时，并不会明确地知道自己在讲什么，只是综合了各种感觉而得到的模糊结论。当然，在平时生活中，并不用每件事物都得探究到底，只是当某个事物是喜爱的、想赖以维生的，或者是两者综合，以为喜欢某个事物，因而想要进一步赖以维生，这个时候就不能只靠个模糊结论来搪塞下去了。

当你拿起这本书，表示选择了 Java 这门程序设计语言，为什么呢？因为 Java 可以写程序。可以写程序的语言很多啊！因为可以写手机 APP？那为什么不选 Objective-C 或 Swift 呢？因为听说业界很缺？喔！缺的是哪个工作性质的职位？手机……好吧！再问下去，可能有人只是被说服参加了三个月的 APP 补习课程，只好硬着头皮继续学下去了……

Java 本身是门程序设计语言，本身就有设计的成分在里头，基于面向对象典范，后来有了一些函数式典范的影子。Java 不是门简洁的语言，然而为了解决这方面问题，近来在语言设计上有了不少简化语法，把 Java 用在许多层面，Java 面对了为模块化制订标准的需求……

然而解决问题并不能只靠程序语言，面对不同的问题，需要设计各种流程，也有不同的数据结构设计。对于更复杂的问题，得靠更有效率的算法。当项目有一定规模，有弹性的架构设计是必要的。为了掌握程序的行为，就得设计可测试的程序；想要避免被入侵，必须将安全上的设计纳入考虑；当庞大的数据迎面而来，平行化方案的设计可能就得出现了……

设计是用来解决问题的！你喜欢或讨厌一种设计，代表它在解决问题上是否顺你的心、得你的意，从而认定一个设计是否优雅。

版权所有，严禁传播！网络传播本PDF！

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：

1、书籍版权归著者和出版社所有

2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用

4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负

5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF

6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

良葛格 Java 学习笔记

既然已经选择了 Java，那表示你得接受它的设计了。那么接下来的问题，就在于怎么使用 Java 来表达你的设计了。不过，这也得真的有能够表达的设计，你有能表达的算法设计吗？数据结构设计？测试上的设计？架构上的设计？安全上的设计？平行处理的设计？……

过去、现在或未来，你写的程序中，真的有“设计”的成分在吗？

林信良
2017 年 12 月

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF！ 版权所有，严禁传播！

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

导 读

这份导读让你可以更了解如何使用本书。

新旧版差异

就目录上来说，你可以看出的差异是，上一版为 18 个章节，新版为 19 个章节，第 19 章“深入模块化”是新的章节，也是 JDK9 最重要的新增功能。然而，认识模块化最好的方式是从实际的例子着手，因此第 1~18 章，全部的范例都是采用模块项目构建。而在各章说明时若有需要，也适时地带入了常用的模块化概念。第 19 章一开始则是整理前 18 个章节遇到过的模块化介绍，然后紧接着深入探讨模块化。

当然，照例要谈一些 JDK9 的其他新增功能，散落在各章节中适当的地方介绍。如果发现页侧有 图标，表示提及 JDK9 新功能，本书还提供了 JDK9 新功能快速查询目录。

虽然程序常用来处理计算，然而许多开发者对数字处理其实认识不多，因而第 15 章通用 API 增加了数字处理的内容；为了认识 JDK9 Stack-Walking API，读者有必要先认识如何取得并运用 StackTraceElement 来进行堆栈追踪，因而在第 15 章还增加了一个小节来介绍堆栈追踪，对于读者了解应用程序行为，或者是处理 Bug，应该会有所帮助。

在第 2 章介绍了模块化之后，范例项目便基于模块化设计了，并使用了 JDK9 的语法增强或改进部分程序代码；在 9.1.6 节介绍 Lambda 之后，为了提高可读性，使用 Lambda 相关语法或 API 来操作程序范例。

旧版有个附录 B“窗口程序设计”，在新版中删掉了，这表示了 Java 在窗口程序这块的地位。当然，Java 有 Java FX 这项技术，如果仍希望使用 Java 进行窗口程序设计，可以寻找 Java FX 的专书。附录 B 虽然不在了，不过范例项目的程序代码留着作为第 19 章的练习，读者可以自行研读原始码，并试着将之模块化。如果一定需要点说明，可以参考旧版《Java SE 6 技术手册》第 19 章说明：

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF！ 版权所有，严禁传播！

- github.com/JustinSDK/JavaSE6Tutorial/blob/master/docs/CH19.md

字型

本书正文中与程序代码相关的文字，都用固定字体来加以呈现，以与一般名词相区别。例如，JDK 是一般名词，而 String 则是程序代码相关文字，所以使用了固定字体以区分。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用
版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

程序范例

读者可以在以下网址下载本书的范例：

- http://www.tupwk.com.cn/downpage
- http://books.gotop.com.tw/v_ACL052100

本书许多范例都使用完整程序操作来展现，如看到以下程序代码示范时：

ClassObject Guess.java

```
package cc.openhome;

import java.util.Scanner;  ←❶ 告诉编译程序接下来想偷懒
import static java.lang.System.out;

public class Guess {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); ←❷ 建立 Scanner 实例
        int number = (int) (Math.random() * 10);
        int guess;

        do {
            System.out.print("猜数字 (0 ~ 9) :");
            guess = scanner.nextInt(); ←❸ 取得下一个整数
        } while(guess != number);

        out.println("猜中了...XD");
    }
}
```

范例开始的左边名称为 ClassObject 表示可以在范例文件的 samples 文件夹的各章节文件夹中找到对应的 ClassObject 项目；而右边名称为 Guess.java 表示可以在项目中找到 Guess.java 文件。如果程序代码中出现标号与提示文字，表示在后续的正文中，会有对应于标号及提示的更详细说明。

原则上，建议读者每个项目范例都能亲手动作撰写，但出于教学时间或操作时间上的考虑，本书有建议进行的练习。如果在范例开始前有个 Lab 图标，例如：

非卖品，仅供非商业用途或交流学习使用
严禁网络传播本PDF！
版权所有，严禁

严禁网络传播！

Lab

Game1 SwordsMan.java

```
package cc.openhome;

public class SwordsMan extends Role {
    public void fight() {
        System.out.println("挥剑攻击");
    }
}
```

表示建议范例动手操作，而且在范例文件的 labs 文件夹中提供了练习项目的基础，打开项目后，完成项目中遗漏或必须补齐的程序代码或设定即可。

如果使用以下程序代码呈现，则表示它是一个完整的程序内容，而不是项目的一部分，这主要用来展现一个完整文档的撰写方法。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用
版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

chapter
导 读

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello!World!");
    }
}
```

如果使用以下程序代码，则表示它是个代码段，主要展现程序撰写时需要特别注意的片段：

```
SwordsMan swordsMan = new SwordsMan();
...
System.out.printf("剑士 (%s, %d, %d)%n", swordsMan.getName(),
        swordsMan.getLevel(), swordsMan.getBlood());
Magician magician = new Magician();
...
System.out.printf("魔法师 (%s, %d, %d)%n", magician.getName(),
        magician.getLevel(), magician.getBlood());
```

提示框

在本书中会出现以下提示框：

提示 »» 针对课程中所提到的观点，提供了一些额外的资源或思考方向，暂时忽略这些提示对课程的影响，但有时间的话，针对这些提示做阅读、思考或讨论是有帮助的。

非卖品，仅供非商业用
严禁网络传播本PDF

非商业用途或交流学习使用

注意 >>> 针对课程中所提到的观点，以提示框方式特别呈现出必须注意的一些使用方式、陷阱或避开问题的方法，看到这个提示框时请读者集中精神阅读。

版权所有，严禁传播！

附录

范例文件包括本书中的所有范例，提供 NetBeans 范例项目，附录 A 用于说明如何使用这些范例项目。

联系作者

若有本书勘误反馈等相关书籍问题，可通过网站与作者联系。网址如下：

http://openhome.cc

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权相关注意事项：

1、书籍版权归著者和出版社所有

2、本PDF来自于各个广泛的信息平台，经过整理而成

3、本PDF仅限用于非商业用途或者个人交流研究学习使用

4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负

5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF

6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅

7、请于下载PDF后24小时内研究使用并删掉本PDF

目　录

Chapter 1　Java 平台概论 ····· 1
1.1　Java 不只是语言 ····· 2
1.1.1　前世今生 ····· 2
1.1.2　三大平台 ····· 5
1.1.3　JCP 与 JSR ····· 6
1.1.4　Oracle JDK 与 OpenJDK ····· 7
1.1.5　建议的学习路径 ····· 8
1.2　JVM/JRE/JDK ····· 12
1.2.1　什么是 JVM ····· 12
1.2.2　区分 JRE 与 JDK ····· 14
1.2.3　下载、安装 JDK ····· 15
1.2.4　认识 JDK 安装内容 ····· 18
1.3　重点复习 ····· 19

Chapter 2　从 JDK 到 IDE ····· 21
2.1　从 Hello World 开始 ····· 22
2.1.1　撰写 Java 原始码 ····· 22
2.1.2　PATH 是什么 ····· 24
2.1.3　JVM(java)与 CLASSPATH ····· 27
2.1.4　编译程序(javac)与 CLASSPATH ····· 29
2.2　管理原始码与位码文档 ····· 30
2.2.1　编译程序(javac)与 SOURCEPATH ····· 30
2.2.2　使用 package 管理类 ····· 32
2.2.3　使用 import 偷懒 ····· 34
2.3　初识模块平台系统 ····· 36
2.3.1　JVM(java)与 module-path ····· 37
2.3.2　编译程序(javac)与 module-path ····· 39
2.3.3　编译程序(javac)与 module-source-path ····· 40
2.4　使用 IDE ····· 41
2.4.1　IDE 项目管理基础 ····· 41
2.4.2　使用了哪个 JRE ····· 47
2.4.3　类文档版本 ····· 49
2.5　重点复习 ····· 51

Chapter 3　基础语法 ····· 53
3.1　类型、变量与运算符 ····· 54
3.1.1　类型 ····· 54
3.1.2　变量 ····· 57
3.1.3　运算符 ····· 60
3.1.4　类型转换 ····· 65
3.2　流程控制 ····· 68
3.2.1　if...else 条件式 ····· 68
3.2.2　switch 条件式 ····· 70
3.2.3　for 循环 ····· 72
3.2.4　while 循环 ····· 73
3.2.5　break、continue ····· 74
3.3　重点复习 ····· 76
3.4　课后练习 ····· 77

Chapter 4　认识对象 ····· 78
4.1　类与对象 ····· 79
4.1.1　定义类 ····· 79
4.1.2　使用标准类 ····· 81
4.1.3　对象指定与相等性 ····· 84
4.2　基本类型打包器 ····· 86
4.2.1　打包基本类型 ····· 86
4.2.2　自动装箱、拆箱 ····· 87
4.2.3　自动装箱、拆箱的内幕 ····· 88
4.3　数组对象 ····· 90

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

4.3.1 数组基础……91
4.3.2 操作数组对象……93
4.3.3 数组复制……98
4.4 字符串对象……100
4.4.1 字符串基础……101
4.4.2 字符串特性……103
4.4.3 字符串编码……107
4.5 查询 Java API 文件……108
4.6 重点复习……113
4.7 课后练习……114

Chapter 5 对象封装……116
5.1 何谓封装……117
5.1.1 封装对象初始流程……117
5.1.2 封装对象操作流程……119
5.1.3 封装对象内部数据……121
5.2 类语法细节……123
5.2.1 public 权限修饰……123
5.2.2 关于构造函数……125
5.2.3 构造函数与方法重载……126
5.2.4 使用 this……128
5.2.5 static 类成员……130
5.2.6 不定长度自变量……135
5.2.7 内部类……136
5.2.8 传值调用……138
5.3 重点复习……140
5.4 课后练习……141

Chapter 6 继承与多态……142
6.1 何谓继承……143
6.1.1 继承共同行为……143
6.1.2 多态与 is-a……147
6.1.3 重新定义行为……150
6.1.4 抽象方法、抽象类……153
6.2 继承语法细节……154
6.2.1 protected 成员……154
6.2.2 重新定义的细节……156
6.2.3 再看构造函数……157
6.2.4 再看 final 关键字……159
6.2.5 java.lang.Object……160
6.2.6 关于垃圾收集……165
6.2.7 再看抽象类……167
6.3 重点复习……169
6.4 课后练习……170

Chapter 7 接口与多态……171
7.1 何谓接口……172
7.1.1 接口定义行为……172
7.1.2 行为的多态……175
7.1.3 解决需求变化……178
7.2 接口语法细节……183
7.2.1 接口的默认……183
7.2.2 匿名内部类……187
7.2.3 使用 enum 枚举常数……190
7.3 重点复习……192
7.4 课后练习……193

Chapter 8 异常处理……194
8.1 语法与继承架构……195
8.1.1 使用 try、catch……195
8.1.2 异常继承架构……197
8.1.3 要抓还是要抛……202
8.1.4 贴心还是造成麻烦……205
8.1.5 认识堆栈追踪……206
8.1.6 关于 assert……210
8.2 异常与资源管理……213
8.2.1 使用 finally……213
8.2.2 自动尝试关闭资源……215
8.2.3 java.lang.AutoCloseable 接口……217
8.3 重点复习……221
8.4 课后练习……222

Chapter 9 Collection 与 Map……223
9.1 使用 Collection 收集对象……224
9.1.1 认识 Collection 架构……224
9.1.2 具有索引的 List……225
9.1.3 内容不重复的 Set……228

严禁网络传播本PDF，违者责任自负！

9.1.4 支持队列操作的 Queue …… 232
9.1.5 使用泛型 …… 234
9.1.6 简介 Lambda 表达式 …… 238
9.1.7 Interable 与 Iterator …… 240
9.1.8 Comparable 与 Comparator …… 243
9.2 键值对应的 Map …… 248
9.2.1 常用 Map 操作类 …… 249
9.2.2 访问 Map 键值 …… 252
9.3 不可变的 Collection 与 Map …… 255
9.3.1 浅谈不可变特性 …… 255
9.3.2 Collections 的 unmodifiableXXX() 方法 …… 256
9.3.3 List、Set、Map 的 of() 方法 …… 258
9.4 重点复习 …… 260
9.5 课后练习 …… 262
Chapter 10 输入/输出 …… 263
10.1 InputStream 与 OutputStream …… 264
10.1.1 串流设计的概念 …… 264
10.1.2 串流继承架构 …… 266
10.1.3 串流处理装饰器 …… 269
10.2 字符处理类 …… 273
10.2.1 Reader 与 Writer 继承架构 …… 274
10.2.2 字符处理装饰器 …… 275
10.3 重点复习 …… 277
10.4 课后练习 …… 278
Chapter 11 线程与并行 API …… 279
11.1 线程 …… 280
11.1.1 线程简介 …… 280
11.1.2 Thread 与 Runnable …… 282
11.1.3 线程生命周期 …… 284
11.1.4 关于 ThreadGroup …… 290
11.1.5 synchronized 与 volatile …… 292
11.1.6 等待与通知 …… 301
11.2 并行 API …… 305
11.2.1 Lock、ReadWriteLock 与 Condition …… 305
11.2.2 使用 Executor …… 313
11.2.3 并行 Collection 简介 …… 323
11.3 重点复习 …… 326
11.4 课后练习 …… 327
Chapter 12 Lambda …… 328
12.1 认识 Lambda 语法 …… 329
12.1.1 Lambda 语法概览 …… 329
12.1.2 Lambda 表达式与函数接口 …… 332
12.1.3 Lambda 遇上 this 与 final …… 334
12.1.4 方法与构造函数参考 …… 336
12.1.5 接口默认方法 …… 338
12.2 Functional 与 Stream API …… 343
12.2.1 使用 Optional 取代 null …… 343
12.2.2 标准 API 的函数接口 …… 345
12.2.3 使用 Stream 进行管道操作 …… 348
12.2.4 进行 Stream 的 reduce 与 collect …… 351
12.2.5 关于 flatMap() 方法 …… 356
12.2.6 Stream 相关 API …… 359
12.2.7 JDK9 Optional 与 Stream 增强 …… 360
12.3 Lambda、平行化与异步处理 …… 362
12.3.1 Stream 与平行化 …… 362
12.3.2 Arrays 与平行化 …… 366

12.3.3 CompletableFuture 非同步处理……367
12.3.4 JDK9 CompletableFuture 增强……369
12.4 重点复习……370
12.5 课后练习……371
Chapter 13 时间与日期……372
13.1 认识时间与日期……373
13.1.1 时间的度量……373
13.1.2 年历简介……374
13.1.3 认识时区……375
13.2 认识 Date 与 Calendar……376
13.2.1 时间轴上瞬间的 Date……376
13.2.2 格式化时间日期的 DateFormat……377
13.2.3 处理时间日期的 Calendar……379
13.2.4 设定 TimeZone……382
13.3 新时间日期 API……383
13.3.1 机器时间观点的 API……383
13.3.2 人类时间观点的 API……385
13.3.3 对时间的运算……387
13.3.4 年历系统设计……389
13.4 重点复习……390
13.5 课后练习……391
Chapter 14 NIO 与 NIO2……393
14.1 认识 NIO……394
14.1.1 NIO 概述……394
14.1.2 Channel 架构与操作……395
14.1.3 Buffer 架构与操作……396
14.2 NIO2 文件系统……398
14.2.1 NIO2 架构……398
14.2.2 操作路径……399
14.2.3 属性读取与设定……401
14.2.4 操作文档与目录……404
14.2.5 读取、访问目录……406
14.2.6 过滤、搜索文档……410
14.3 重点复习……412
14.4 课后练习……413
Chapter 15 通用 API……414
15.1 日志……415
15.1.1 日志 API 简介……415
15.1.2 指定日志层级……417
15.1.3 使用 Handler 与 Formatter……419
15.1.4 自定义 Handler、Formatter 与 Filter……420
15.1.5 使用 logging.properties……422
15.2 国际化基础……423
15.2.1 使用 ResourceBundle……423
15.2.2 使用 Locale……424
15.3 规则表示式……426
15.3.1 规则表示式简介……426
15.3.2 Pattern 与 Matcher……433
15.4 处理数字……435
15.4.1 使用 BigInteger……435
15.4.2 使用 BigDecimal……437
15.4.3 数字的格式化……439
15.5 再谈堆栈追踪……441
15.5.1 获取 StackTraceElement……441
15.5.2 JDK9 的 Stack-Walking API……443
15.6 重点复习……447
15.7 课后练习……448
Chapter 16 整合数据库……449
16.1 JDBC 入门……450
16.1.1 JDBC 简介……450
16.1.2 连接数据库……454
16.1.3 使用 Statement、ResultSet……459
16.1.4 使用 PreparedStatement、CallableStatement……464

16.2 JDBC 进阶 ····· 468
16.2.1 使用 DataSource 取得联机 ····· 468
16.2.2 使用 ResultSet 卷动、更新数据 ····· 471
16.2.3 批次更新 ····· 473
16.2.4 Blob 与 Clob ····· 474
16.2.5 交易简介 ····· 474
16.2.6 metadata 简介 ····· 481
16.2.7 RowSet 简介 ····· 484
16.3 重点复习 ····· 486
16.4 课后练习 ····· 487

Chapter 17 反射与类加载器 ····· 489
17.1 运用反射 ····· 490
17.1.1 Class 与.class 文档 ····· 490
17.1.2 使用 Class.forName() ····· 492
17.1.3 从 Class 获得信息 ····· 494
17.1.4 从 Class 建立对象 ····· 496
17.1.5 操作对象方法与成员 ····· 499
17.1.6 动态代理 ····· 501
17.1.7 当反射遇上模块 ····· 505
17.1.8 使用 ServiceLoader ····· 511
17.2 了解类加载器 ····· 513
17.2.1 JDK9 类加载器层级 ····· 513
17.2.2 建立 ClassLoader 实例 ····· 516
17.3 重点复习 ····· 517
17.4 课后练习 ····· 519

Chapter 18 自定义泛型、枚举与注释 ····· 520
18.1 自定义泛型 ····· 521
18.1.1 使用 extends 与? ····· 521
18.1.2 使用 super 与? ····· 525
18.2 自定义枚举 ····· 528
18.2.1 了解 java.lang.Enum 类 ····· 528
18.2.2 enum 高级运用 ····· 531
18.3 关于注释 ····· 536
18.3.1 常用标准注释 ····· 536
18.3.2 自定义注释类型 ····· 540
18.3.3 执行时期读取注释信息 ····· 545
18.4 重点复习 ····· 548
18.5 课后练习 ····· 549

Chapter 19 深入模块化 ····· 550
19.1 运用模块 ····· 551
19.1.1 模块的种类 ····· 551
19.1.2 requires、exports 与 opens 细节 ····· 554
19.1.3 修补模块 ····· 557
19.1.4 放宽模块封装与依赖 ····· 558
19.2 模块 API ····· 560
19.2.1 使用 Module ····· 560
19.2.2 使用 ModuleDescriptor ····· 562
19.2.3 浅谈 ModuleLayer ····· 562
19.3 打包模块 ····· 564
19.3.1 使用 jar 打包 ····· 564
19.3.2 使用 jmod 打包 ····· 566
19.3.3 使用 jlink 建立执行时期映像 ····· 568
19.4 重点复习 ····· 569
19.5 课后练习 ····· 570

Appendix ····· 571
A.1 项目环境配置 ····· 572
A.2 打开案例 ····· 572

Java SE 9 新功能索引

Java SE 9 后的特性版本时间轴变动 …… 15
JDK 9 文档实体布局变动 …… 18
初探模块平台系统 …… 36
javac 新增-release 参数 …… 50
支持 Unicode 8.0 …… 54
内建 jshell …… 56
Java API 文件支持搜索功能 …… 112
StackTraceElement 新增方法 …… 207
Try-with-resources 语法改进 …… 217
定义匿名类别时的泛型语法改进 …… 239
List、Set、Map 新增 of( )方法 …… 258
接口支持定义 private 方法 …… 340
Collectors 新增 filtering( )方法 …… 356
Collectors 新增 flatMapping( )方法 …… 359
Optional 与 Stream 增强 …… 360
CompletableFuture 增强 …… 369
支持 UTF-8 编码的.properities 文件 …… 425
Stack-Walking API …… 443
反射与类加载器机制 …… 490
@Deprecated 增强 …… 537
ElementType 新增 MODULE …… 545
深入模块化 …… 551

Chapter 1 Java 平台概论

学习目标

- Java 版本迁移简介
- 认识 Java SE、Java EE、Java ME
- 认识 JDK 规范与操作
- 了解 JVM、JRE 与 JDK
- 下载与安装 JDK

1.1 Java不只是语言

从1995年至今，Java已经过了20多个年头，经过这些年的改进，正如本节标题所示，Java已不仅是个程序语言，也代表了解决问题的平台(Platform)，更代表了原厂、各个厂商、社群、开发者与用户沟通的成果。若仅以程序语言的角度来看待Java，正如冰山一角，你仅看到Java身为程序语言的一部分，而没看到Java身为程序语言之外，更可贵也更为庞大的资源。

1.1.1 前世今生

一个语言的诞生有其目的，因为这个目的而成就了该语言的主要特性。探索Java的历史演变，对于掌握Java特性与各种可用资源，有很大帮助。

1. Java诞生

Java最早是Sun公司绿色项目(Green Project)中撰写Star7应用程序的程序语言，当时的名称并不是Java，而是Oak。

绿色项目始于1990年12月，由Patrick Naughton、Mike Sheridan与James Gosling(James Gosling被尊称为Java之父)主持，目的是希望构筑出下一波计算机应用趋势并加以掌握，他们认为下一波计算机应用趋势会集中在消费性数字产品(就像现在的PDA、手机等消费性电子商品)的使用上。1992年9月3日，Green Team项目小组展示了Star7手持设备，这个设备具备无线网络连接、5寸LCD彩色屏幕、PCMCIA接口等功能，而Oak在绿色项目中的目的，是用来撰写Star7上应用程序的程序语言。

Oak名称的由来，是因为James Gosling的办公室窗外有一棵橡树(Oak)，就取了这个名称。但后来发现Oak已经被注册了，工程师们边喝咖啡边讨论着新名称，最后灵机一动而改名为Java。

Java本身有许多为了节省资源而作的设计，如动态加载类别文档、字符串池(String Pool)等特性，这是因为Java一开始就是为了消费性数字产品而设计，而这类小型装置通常有着有限内存与运算资源。

全球信息网(World Wide Web)兴起，Java Applet成为网页互动技术的代表。1993年第一个全球信息网浏览器Mosaic诞生，James Gosling认为互联网与Java的一些特性不谋而合，利用Java Applet在浏览器上展现互动性媒体，在当时而言，对视觉感官是一种革命性的颠覆。Green Team仿照Mosaic开发出以Java技术为基础的浏览器WebRunner(原名为BladeRunner)，后来改名为HotJava。虽然HotJava只是一个展示性产品，但它使用Java Applet展现的多媒体效果立即吸引了许多人的注意，图1.1所示即为JDK所附的Java Applet范例。

1995年5月23日(这一天被公认为Java的诞生日)，正式将Oak改名为Java，Java Development Kits(当时JDK的全名)1.0a2版本正式对外发表。到1996年，Netscape Navigator 2.0也正式支持Java，Microsoft Internet Explorer也开始支持Java。从此，Java在互联网的世界中逐渐风行起来。虽然Star7产品并不被当时消费性市场接受，绿色项目面临被裁撤的命运，然而全球信息网的兴起却给了Java新的生命与舞台。

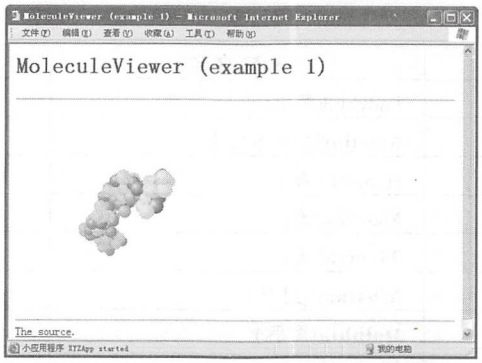


图 1.1 JDK 所附的 Java Applet 范例(JDK 文件夹\demo\applets\MoleculeViewer\example1.html)

2. 版本演进

随着 Java 越来越受到瞩目，Sun 在 1998 年 12 月 4 日发布 Java 2 Platform，简称 J2SE 1.2。Java 开发者版本一开始是以 Java Development Kit 名称发表，简称 JDK，而 J2SE 则是平台名称，包含了 JDK 与 Java 程序语言。

Java 平台标准版约以两年为周期推出重大版本更新，1998 年 12 月 4 日发表 J2SE 1.2，2000 年 5 月 8 日发表 J2SE 1.3，2002 年 2 月 13 日发表 J2SE 1.4，Java 2 这个名称也从 J2SE 1.2 开始沿用至各个版本。

2004 年 9 月 29 日发表的 Java 平台标准版的版号不是 1.5，而直接跳到 5.0，称为 J2SE 5.0，这是为了彰显这个版本与之前版本有极大不同，如语法上的简化、增加泛型(Generics)、枚举(Enum)、注释(Annotation)等重大功能。

2006 年 12 月 11 日发表的 Java 平台标准版，除了版本号之外，名称也有了变化，称为 Java Platform, Standard Edition 6，简称 Java SE 6。JDK6 全名则称为 Java SE Development Kit 6，也就是不再像以前 Java 2 带有 2 这个号码，版本号 6 或 1.6.0 都使用，6 是产品版本(Product Version)，而 1.6.0 是开发者版本(Developer Version)。

大部分的 Java 标准版平台都会取个代码名称(Code Name)，例如 J2SE 5.0 的代码名称为 Tiger(老虎)，为了引人注目，在发表会上还真的抱了一只小白老虎出来作为噱头，而许多书的封面也相应地放上老虎的图片。有关 JDK 代码名称与发布日期，如表 1.1 所示。

表 1.1 Java 版本、代码名称与发布日期

版 本	代 码 名 称	发 布 日 期
JDK 1.1.4	Sparkler(烟火)	1997/9/12
JDK 1.1.5	Pumpkin(南瓜)	1997/12/3
JDK 1.1.6	Abigail(圣经故事人物名称)	1998/4/24
JDK 1.1.7	Brutus(罗马政治家名称)	1998/9/28
JDK 1.1.8	Chelsea(足球俱乐部名称)	1999/4/8
J2SE 1.2	**Playground(游乐场)**	**1998/12/4**
J2SE 1.2.1	无	1999/3/30
J2SE 1.2.2	Cricket(蟋蟀)	1999/7/8
J2SE 1.3	**Kestrel(红隼)**	**2000/5/8**

(续表)

版　本	代码名称	发布日期
J2SE 1.3.1	Ladybird(瓢虫)	2001/5/17
J2SE 1.4.0	**Merlin(魔法师名称)**	**2002/2/13**
J2SE 1.4.1	Hopper(蚱蜢)	2002/9/16
J2SE 1.4.2	Mantis(螳螂)	2003/6/26
J2SE 5.0	**Tiger(老虎)**	**2004/9/29**
Java SE 6	**Mustang(野马)**	**2006/12/11**
Java SE 7	**Dolphin(海豚)**	**2011/7/28**
Java SE 8	无	**2014/3/18**
Java SE 9	无	**2017/9/21**

提示 »» 撰写本书时，表 1.1 参考的数据源网址如下(其中列出至 J2SE 5.0)：
http://www.oracle.com/technetwork/java/javase/codenames-136090.html

3. 江山易主

之前谈过，Java 约以两年为周期推出重大版本更新，正如表 1.1 所示，J2SE 1.2、J2SE 1.3、J2SE 1.4.0、J2SE 5.0、Java SE 6 推出的时间间隔，差不多都是两年。然而从 Java SE 6 之后，Java 开发人员足足等了四年多，才等到新版本的推出，不禁让人想问：Java 怎么了？

原因有许多，Java SE 7 对新版本的规划摇摆不定，涵盖许多不易实现的新特性，加上 Sun 一直营收低迷不振，影响了新版本的推动。新版本推出日期承诺不断推迟，从 2009 年推迟至 2010 年初，又突然宣布将加入原本不愿划入 Java SE 7 的 Closure 语法，并将推出日期推迟至 2010 年底。2010 年年中传出 IBM 与 Sun 密谈并购失败，没隔几日，即爆出 Oracle 宣布并购 Sun，Java 也正式成为 Oracle 所属。

并购会带来一连串的组织重整，导致 Java SE 7 推出日期再度推迟，为了对停滞不前的 Java 注入活水，决定先将现有已实现或较易实现的特性放入 Java SE 7 中，将未定方案或较难实现的特性放入 Java SE 8 中(比如 Lambda)，2010 年底 JCP(Java Community Process，稍后即会说明这个组织是什么)终于通过了 Java SE 7 与 Java SE 8 的规划地图(Roadmap)，并预定于 2011 年 7 月左右推出 Java SE 7，这次总算没有推迟，Java SE 7 正式于 2011 年 7 月 28 日发布。

Java SE 8 实际上也是一波三折，原定应于 2013 年发布，却因为接二连三爆出的 Java 安全漏洞，迫使 Java 开发团队决定先行检查修补 Java 安全问题，几经延后，最后确定发布 Java SE 8 的时间为 2014 年 3 月 18 日。

实际上，当初为了及早推出 Java SE 7，被推迟至 Java SE 8 的重大特性还有 Jigsaw，也就是 Java 模块平台系统(Java Platform Module System)。然而当时为了修补 Java 安全问题已经使得 Java SE 8 多次延迟，为了能如期推出 Java SE 8，Java 模块平台系统并没有在 Java SE 8 中实现，而被放到了 Java SE 9 中并成为重大特性之一。虽然如此，推出过程中又因为开发不及，以及发生了 JCP 执行委员会(JCP Executive Committee)曾经投票否决了 Java 模块平台系统等因素，使得 Java SE 9 发布日期多次推后，终于在 2017 年 9 月 21 日正式发布。

提示 »» 可以在下面的页面中，看到 Java 模块平台系统一度被否决的投票结果：
http://jcp.org/en/jsr/results?id=5959

1.1.2 三大平台

在 Java 发展的过程中，由于其应用领域越来越广，并逐渐扩展至各级应用软件的开发，Sun 公司在 1999 年 6 月美国旧金山举办的 Java One 大会上，公布了新的 Java 体系架构。该架构根据不同级别的应用开发区分了不同的应用版本：J2SE(Java 2 Platform, Standard Edition)、J2EE(Java 2 Platform, Enterprise Edition)与 J2ME(Java 2 Platform, Micro Edition)。

J2SE、J2EE 与 J2ME 是当时的名称，由于 Java SE 6 后 Java 不再带有 2 这个号码，J2SE、J2EE 与 J2ME 分别被正名为 Java SE、Java EE 与 Java ME。

提示 »» 尽管 Sun 从 2006 年底，就将三大平台正名为 Java SE、Java ME 与 Java EE，但时至今日，许多人的习惯并没有改过来，J2SE、J2ME 与 J2EE 这些名词还是有很多人用。

1. Java SE

Java 是各应用平台的基础，想要学习其他的平台应用，必先了解 Java SE 以奠定基础，而 Java SE 也正是本书主要的介绍对象。

图 1.2 所示是整个 Java SE 的组成概念图。

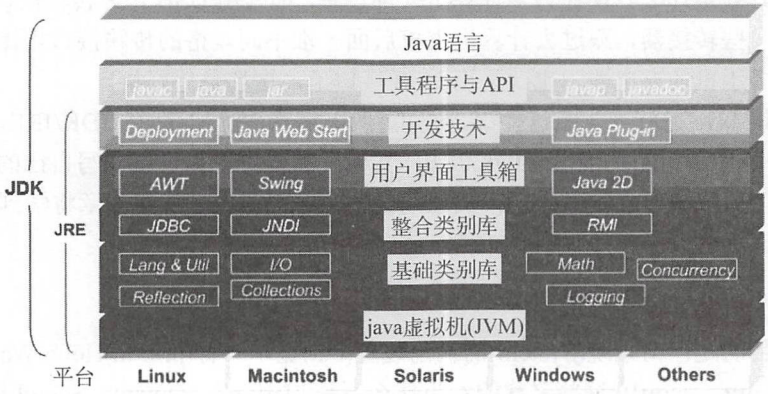


图 1.2 Java SE 的组成概念图

Java SE 可以分为四个主要的部分：JVM、JRE、JDK 与 Java 语言。

为了能够运行 Java 撰写好的程序，必须有 Java 虚拟机(Java Virtual Machine, JVM)。JVM 包括在 Java 执行环境(Java SE Runtime Environment, JRE)中，所以要运行 Java 程序，必须先安装 JRE。如果要开发 Java 程序，必须取得 JDK(Java SE Development Kits)，JDK 包括 JRE 及开发过程中需要的一些工具程序，如 javac、java 等工具程序(关于 JRE 及 JDK 的安装与使用方法，会在第 2 章说明)。

Java 语言只是 Java SE 的一部分，除此以外，Java 最重要的就是提供庞大且强大的标准 API，提供字符串处理、数据输入/输出、网络套件、用户窗口接口等功能，可以使用这些 API 作为基础来进行程序开发，无须重复开发功能相同的组件。事实上，在熟悉 Java 语言之后，更多的时候，都是在学习如何使用 Java SE 提供的 API 来完成应用程序。

2. Java EE

Java EE 以 Java SE 为基础，定义了一系列的服务、API、协议等，适用于开发分布式、多层次(Multi-tiered)、以组件为基础、以 Web 为基础的应用程序。整个 Java EE 的体系是相当庞大的，比较为人熟悉的技术是 JSP、Servlet、JavaMail、Enterprise JavaBeans(EJB)等，其中每个

服务或技术都可以使用专书进行说明，并非本书说明的范围。但可以肯定的是，必须在 Java SE 上奠定良好的基础，再来学习 Java EE 的开发。

3. Java ME

Java ME 是 Java 平台版本中最小的应用程序，目的是作为小型数字设备上开发及部署应用程序的平台，如消费性电子产品或嵌入式系统等，其中最为人熟悉的设备如手机、PDA、股票机等。可以使用 Java ME 来开发出这些设备上的应用程序，如 Java 游戏、股票相关程序、记事程序、日历程序等。

1.1.3 JCP 与 JSR

Java 不仅是程序语言，还是标准规范。

先来看看没有标准会产生什么问题？我们的身边有些东西没有标准，例如手机充电器，不同厂商的手机，充电器也不相同，家里面一堆充电器互不相容，换个手机，充电器就不能用的情况，相信你我都有过。

有标准的好处是什么？现在许多计算机外部设备，都采用 USB 作为传输接口，这让计算机中不用再接上一些转接器，跟过去计算机主机后面一堆不同规格的传输接口相比，实在方便了不少，现在许多手机的充电器，也改采用 USB 接口了，这真是件好事。

回头来谈谈 Java 是标准规范这件事。你知道吗？编译/执行 Java 的 JDK/JRE，并不只有 Sun 才能实现，IBM 也可以撰写自己的 JDK/JRE，其他厂商或组织也可以撰写自己的 JDK/JRE，你写的 Java 程序，可以执行在这些不同厂商或组织写出来的 JRE 上。第 2 章将学到的第一个 Java 程序，其中会有这么一段程序代码：

```
System.out.println("Hello World");
```

这行程序目的是：请系统(System)的输出装置(out)显示一行(println)Hello World。是谁决定使用 System、out、println 这些名称的？为什么不是 Platform、Output、ShowLine 这些名称？如果 Sun 使用 System、out、println 这些名称，而 IBM 使用了 Platform、Output、ShowLine 这些名称，用 Sun 的 JDK 写的程序，就不能执行在 IBM 的 JRE 上，那 Java 最基本的特性之一“跨平台”，就根本无法实现了。

Java 由 Sun 创造，为了让对 Java 感兴趣的厂商、组织、开发者与用户参与定义 Java 未来的功能与特性，Sun 公司于 1998 年组成了 JCP(Java Community Process)，这是一个开放性国际组织，目的是让 Java 的演进由 Sun 非正式地主导，成为全世界数以百计代表成员公开监督的过程。

任何想要提议加入 Java 的功能或特性，必须以 JSR(Java Specification Requests)正式文件的方式提交，JSR 必须经过 JCP 执行委员会(Executive Committee)投票通过，方可成为最终标准文件，有兴趣的厂商或组织可以根据 JSR 实现产品。

若 JSR 成为最终文件后，必须根据 JSR 做出免费且开发原始码的参考实现，称为 RI(Reference Implementation)，并提供 TCK(Technology Compatibility Kit)作为技术兼容测试工具箱，方便其他想根据 JSR 实现产品的厂商或组织参考与测试兼容性。JCP、JSR、RI 与 TCK 的关系，如图 1.3 所示。

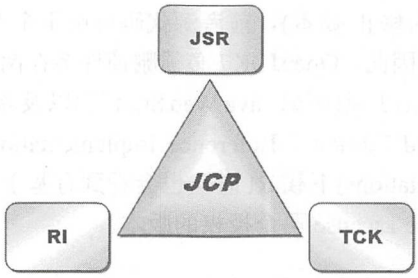


图 1.3　JCP、JSR、RI 与 TCK

提示 »» JCP 官方网站为 http://jcp.org。

现在无论 Java SE、Java EE 还是 Java ME，都是业界共同制定的标准，每个标准背后都隐藏着业界所面临的一些问题，他们期待使用 Java 来解决问题，认为应该有某些组件、特性、应用程序编程接口等，来解决这些问题，因而制定 JSR 作为正式标准规范文件，不同的技术解决方案标准规范会给予一个编号。

在 JSR 规范的标准之下，各厂商可以各自操作成品，所以同一份 JSR，可以有不同厂商的操作产品。以 Java SE 为例，对于身为开发人员，或使用 Java 开发产品的公司而言，只要使用兼容于标准的 JDK/JRE 开发产品，就可以执行、兼容在标准的 JRE 上，而不用担心跨平台的问题。

Java SE 9 的主要规范是在 JSR 379 文件之中，而 Java SE 9 平台中的特定技术，则规范于特定的 JSR 文件之中。若有读者对这些文件有兴趣，可以参考 JSR379，网址如下：

http://jcp.org/en/jsr/detail?id=379

提示 »» 想要查询 JSR 文件，只要在 http://jcp.org/en/jsr/detail?id=之后加上文件编号就可以了，例如上面查询 JSR 379 文件网址就是：
http://jcp.org/en/jsr/detail?id=379
JSR 对于 Java 初学者而言过于生涩，但 JSR 文件规范了相关技术应用的功能，将来有能力时，可以试着自行阅读 JSR，这有助于了解相关技术规范的更多细节。

1.1.4　Oracle JDK 与 OpenJDK

在过去，Sun JDK 实现，也就是被 Oracle 收购之后的 Oracle JDK 实现，就是 JDK 的参考实现，有兴趣的厂商或组织也可以根据 JSR 自行实现产品，例如 IBM 就是根据 JSR 实现了自家的 IBM JDK。只有通过 TCK 兼容性测试的实现，才可以使用 Java 这个商标。

提示 »» IBM JDK：http://www.ibm.com/developerworks/java/jdk/

2006 年的 JavaOne 大会上，Sun 宣告对 Java 开放源代码，从 JDK7 b10 开始有了 OpenJDK，并于 2009 年 4 月 15 日正式发布 OpenJDK。Oracle 时代发布的 JDK7 正式版本，指定了 OpenJDK7 为官方参考实现。

1. Oracle JDK7 与 OpenJDK7

与同为开放源代码的 Sun JDK 不同的是，Sun JDK 采用 JRL，而 OpenJDK7 采用 GPL(带

有 GPL linking exception 的修正版本)，前者源代码可用于个人研究使用，但禁止任何商业用途，后者则允许商业使用。因此，OpenJDK7 必须删掉许多在两个授权间有冲突的程序代码，也不包括一些部署(Deployment)工具(例如 Java Web Start 等)以及软件套件(例如 Java DB)等；现在你在 Java Platform, Standard Edition 7 Reference Implementations(或 Java Platform, Standard Edition 9 Reference Implementations)下载 RI 时，也会看到有基于 GNU General Public License version 2 与 Oracle Binary Code License 两个授权的版本。

提示>>> Java Platform, Standard Edition 7 Reference Implementations：
http://jdk.java.net/java-se-ri/7
Java Platform, Standard Edition 9 Reference Implementations：
http://jdk.java.net/java-se-ri/9

由于 OpenJDK7 中有许多程序代码因授权冲突而必须删掉，因此原始的 OpenJDK7 是不完整的，所以无法通过 TCK 兼容测试。如果执行 `java -version`，原始的 OpenJDK7 显示的会是 openjdk version 字样，而不是 java version 字样。

为了解决授权问题，以便在 Fedora 或 Linux 分支中能自由发布 OpenJDK7，Red Hat 于 2007 年发起了 IcedTea 计划。而由于原始的 OpenJDK7 是不完整的，后来 IcedTea 致力于修补 OpenJDK7 使之完备，并通过了 TCK 兼容测试。如果使用 IcedTea 修补过后的 OpenJDK7，执行 `java -version`，就会显示 java version 字样。

2. OpenJDK7 与 OpenJDK6

在 OpenJDK 官方网站，也可以看到 OpenJDK6 的版本，OpenJDK6 并不是 Sun JDK6 的分支，而是将 OpenJDK7 中 JDK7 的特性删掉，使之符合 JDK6 的规范，因而 OpenJDK6 实际上是 OpenJDK7 的分支，OpenJDK6 可以通过 TCK 兼容测试。

Oracle 从 2012 年 7 月以来，就打算结束对 JDK6 的支持，在 2013 年 2 月时发布 JDK6 Update 43 时，宣布这是最后一个免费更新版本(实际上后来还有 Update 45)，希望大家赶快升级至 JDK7。

由于 JDK6 在企业间仍广泛应用，Red Hat 于 2013 年 3 月宣布持有 OpenJDK6 领导权，以便能持续对 OpenJDK6 发现的漏洞与安全问题进行修补。

1.1.5 建议的学习路径

Java 不仅是程序语言，还是标准规范。每个标准代表着厂商面临的问题，代表着解决问题的方案，因此，学习 Java 就等于在面临各式问题如何解决。然而，这么多的问题衍生出如此多的解决方案，对于初学 Java 的人而言，如同面临满载产品的庞大货轮，不知从何开始，也不知将来何去何从。

提示>>> 如果将程序语言比喻为一艘船，会是如何呢？这里有篇有趣的文章：
http://compsci.ca/blog/if-a-programming-language-was-a-boat/

Java 的官方网站提供了一份 Java 技术概念地图(Java Technology Concept Map)的文件，如图 1.4 所示。这是份 PDF 文件，可以在以下网址下载：

http://www.oracle.com/technetwork/topics/newtojava/intro-142494.html

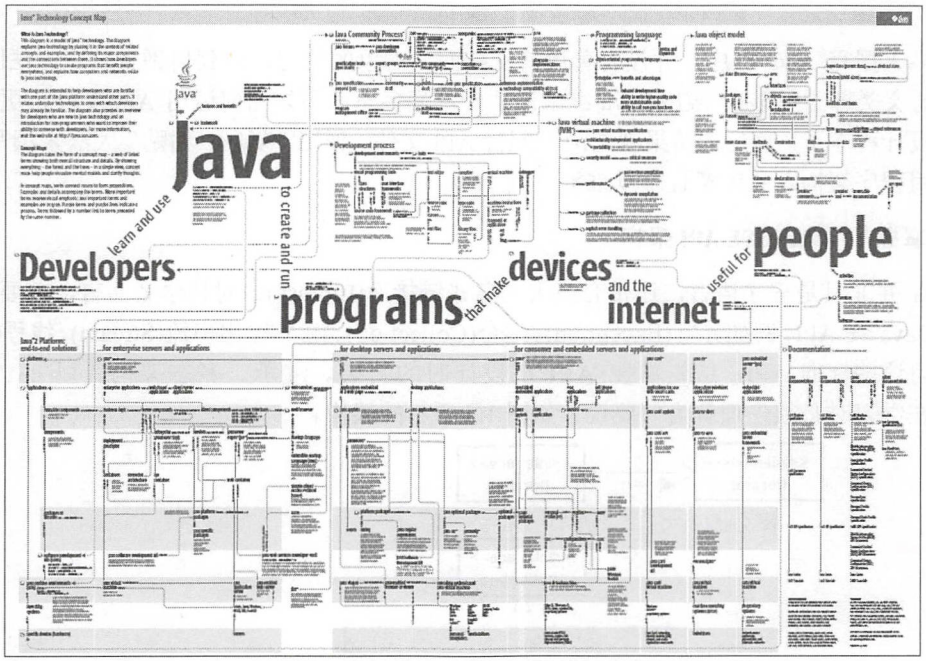


图 1.4　Java 技术概念地图

提示 »»　Oracle 合并 Sun 之后，Java 的官方网站是：
http://java.oracle.com/
连接网络之后，将会被重新链接至：
http://www.oracle.com/technetwork/java/index.html

在这份文件中，密密麻麻地列出了大部分 Java 相关地图与简要说明，也代表了 Java 技术范畴的广泛。然而要从这么庞大的地图中找出一条适合初学 Java 的路线图绝非易事。以下是我基于经验与教学建议的学习路径。

1. 深入了解 JVM/JRE/JDK

许多书籍对于 JVM/JRE/JDK 的说明，通常只用了极短的篇幅介绍，就是在短短几页中，请使用者按书中步骤安装与设定 `PATH`、`CLASSPATH` 后，就开始介绍 Java 程序语言。而许多人到了业界后就开始使用 IDE(Integrated Development Environment)代劳所有 JDK 细节。这么做的结果就是，在 IDE 中遇到与 JDK 相关的问题，就完全不知道如何解决。

JVM/JRE/JDK 并不是用短短几页就可以说明，若没有“JVM 是 Java 程序唯一认识的操作系统，其可执行文件为.class 文档”的重要观念，就无法理解 `PATH` 与 `CLASSPATH` 并非同一层级的环境变量，JDK 中许多指令与选项其实都可以对应至 IDE 中某个设定与操作。对 JVM/JRE/JDK 有了足够的认识，对 IDE 中相关选项就不会有疑问，也不会换个 IDE 就不知所措，或没有 IDE 就无法撰写程序。

2. 理解封装、继承、多态

对于 Java 程序语言，`if...else`、`for`、`while`、`switch` 等流程语法早已是必须熟练的基础。更重要的是，Java 支持面向对象(Object Oriented)，你必须理解面向对象中最重要的封装(Encapsulation)、继承(Inheritance)、多态(Polymorphism)概念，以及如何用 Java 相关语法来

实现。

许多人撰写 Java 程序，并没有善用其支持面向对象的特性，问到何谓封装却无法回答(甚至回答定义类即为封装)，滥用 Java 继承语法，不懂多态而不知如何运用 API 文件，更别说运用多态设计程序了，最后的结果就是沦于死背 API 文件、使用“复制、粘贴”大法来撰写程序，整个应用程序架构杂乱无章且难以维护。

3. 掌握常用 Java SE API 架构

Java 并非只是程序语言，还带有庞大的各式链接库(Library)，对初学者而言，首要是掌握常用的 Java SE API，例如异常(Exception)、集合(Collection)、输入/输出串流(Stream)、线程(Thread)等。学习这些标准 API，应先掌握 API 在设计时的封装、继承、多态架构。以 `Collection` 为例，在学习时必须先理解为何要设计如图 1.5 所示的架构。

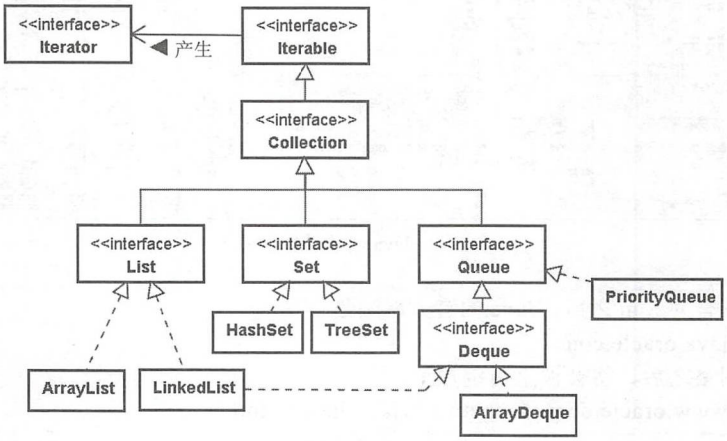


图 1.5　Collection API 架构范例

学习相关链接库或 API，先理解主要架构是有必要的，这样才不会沦于死背 API 或抄写范例的窘境。更进一步地，还可以从 API 中学习到良好设计的观念，有了这样的好习惯，以后对新的 API 或链接库就能更快地掌握如何使用甚至改进。

提示 »» 深入了解 JVM/JRE/JDK，理解封装、继承、多态，掌握常用 Java SE API 架构，这都是本书的说明重点。

4. 学习容器观念

在步入 Java EE 领域之后，经常接触到容器(Container)的观念，许多人完全以 API 层次来使用 Java EE 相关组件，这是不对的。就操作层面来说，容器是执行于 JVM 上的 Java 应用程序；从抽象层面来说，就是应用程序沟通、协调相关资源的系统。

初次接触容器的开发人员会觉得容器很抽象。以实际的例子来说，通常初学者步入 Java EE，会从学习 Servlet/JSP 开始，而 Servlet/JSP 是执行于 Web 容器之中，这是学习容器时不错的开始，你必须知道“Web 容器是 Servlet/JSP 唯一认识的 HTTP 服务器，是使用 Java 撰写的应用程序，运行于 JVM 之上”。如果希望用 Servlet/JSP 撰写的 Web 应用程序可以正常运作，就必须知道 Servlet/JSP 如何与 Web 容器沟通，Web 容器如何管理 Servlet/JSP 的各种对象等问题。

提示 »» 关于 Servlet/JSP 的说明，可参考我撰写的《JSP & Servlet 学习笔记(第 2 版)》(清华大学出版社)，或是 Servlet/JSP 在线文件：

http://openhome.cc/Gossip/ServletJSP/

容器无所不在，Applet 会执行于 Applet 容器中，因此相关资源受到 Applet 容器的管理与限制，Servlet/JSP 执行于 Web 容器中，EJB 执行于 EJB 容器中，Java 应用程序客户端执行于应用程序客户端容器中，如图 1.6 所示。不理解组件如何与容器互动，就无法真正使用或理解组件的行为。

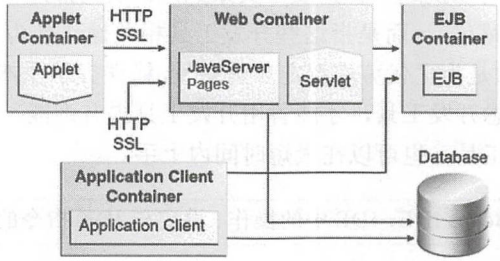


图 1.6 程序组件及其执行容器(摘自 https://docs.oracle.com/javaee/6/tutorial/doc/bnacj.html)

5. 研究开放原始码项目

Java 不仅是程序语言，也是个标准，在共同标准下有不同的运行方式，在 Java 领域的许多操作都是以开放原始码的方式存在，只要你有兴趣，可以下载原始码了解运行方式，从中了解、吸收他人设计、实现产品的技巧或理念。

许多基于 Java 各标准平台发展出来的产品也值得研究，如测试框架(Framework)、Web 框架、持久层(Persistance)框架、对象管理容器等，这些产品补足标准未涵盖之处，各有其设计上的优秀与精良之处，有的已成为 Java 标准之一，它们多以开放原始码的方式存在，让开发人员可以使用、研究甚至参与改进。

提示 »» 想要开始研究开放原始码项目的使用与设计，JUnit 是个不错的开始，可以参考在线文件：

http://openhome.cc/Gossip/JUnit/

6. 学习设计模式与重构

在程序设计上，“经验”是最重要的，在经验传承上，归纳而言，无非就是“如何根据需求做出好的设计”“如何因应需求变化调整现有程序架构”，对于前者，流传下来的设计经验就是设计模式(Design Pattern)，对于后者，流传下来的调整手法就是重构(Refactor)。

“如果我当初就这么设计，现在就不会发生这个问题了！”这种对话应该很熟悉，“当初就这么设计”就是所谓设计模式。“如果我当初先这么改，再那么改，就不会把程序改到烂了！”这种对话也经常听到，“当初先这么改，再那么改”就是所谓重构。

无论好的设计还是不好的设计，都要有经验传承。经验可以口耳相传，可以从原始码中观摩，也可以从书籍或网络上优秀的技术文件中学得。对于初学者，从书籍或网络上优秀的技术文件学习设计模式与重构，是积累经验的快捷方式。

7. 熟悉相关开发工具

除了累积足够的实力与基础，善用工具是必要的，开发工具可以避免烦琐的指令、减少重

复性的操作、提示可用的 API、自动产生程序代码、降低错误的发生，甚至执行各种自动化的测试、报告产生与发送邮件等任务。有些开发人员鄙视开发工具，这是不必要的，两个实力相同的开发人员撰写相同的应用程序，使用良好开发工具的人必然有较高的效率。

在 Java 领域难能可贵的是，存在不少优秀的开发工具，而且多以开放架构、开放原始码的方式存在，如 Eclipse IDE、NetBeans IDE、IntelliJ IDEA 都是相当不错的选择。还可以搭配 Ant 构建工具、Maven 或 Gradle 项目工具等一起使用，大大地提升开发人员的产能。

建议初学 Java 的人，可以挑选一种开发工具来熟悉。所谓熟悉，不是指“下一步要按哪个按钮、接下来要执行哪个菜单”，而是指这些开发工具相关操作是为了代劳你手动执行哪些指令，开发工具中的某些选项是为了代劳你设定哪些变量，错误提示原本是来自 JDK 的什么信息，等等。以这样的过程来熟悉开发工具，才能善用开发工具提升产能，而不是受制于开发工具，这样即便换了另一套开发工具，也可以在最短时间内上手。

提示>>> 本书会使用 NetBeans IDE，IDE 中的操作、设定与 JDK 指令的对照，是本书的重点之一。

1.2 JVM/JRE/JDK

本书一开始曾说，不要只从程序语言的角度来看 Java，这只会看到“冰山一角”。这可以用图 1.7 来印证。

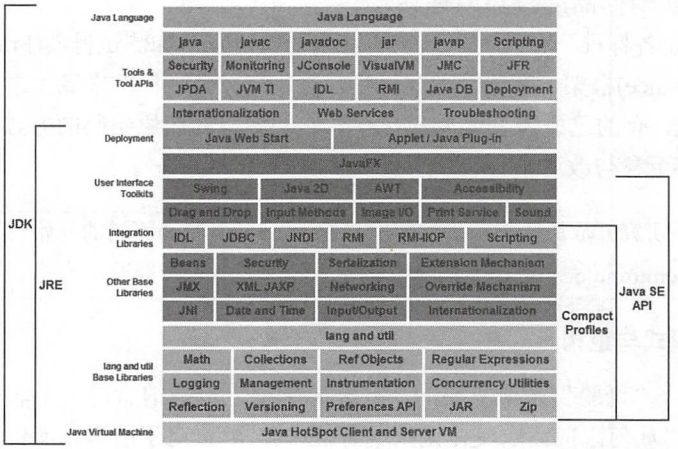


图 1.7 Java 产品概念图(摘自 http://www.oracle.com/technetwork/java/javase/tech/index.html)

如果安装 JDK，就会安装图中所示全部的东西，而 Java Language 只是最左上角一小部分。如果只是用短短几页告诉你如何安装 JDK、设定环境变量，而不解释到底 JVM、JRE 与 JDK 的作用与关系，你觉得够吗？

1.2.1 什么是 JVM

在图 1.7 中，Java Virtual Machine(JVM)会架构在 Linux、Windows、iOS 各种操作系统平台之上。许多 Java 的书都会告诉你，JVM 让 Java 可以跨平台，但是跨平台是怎么一回事，在这之前，得先了解不能跨平台是怎么一回事。

对于计算机而言，只认识一种语言，也就是 0、1 序列组成的机器指令。当使用 C/C++等高级语言撰写程序时，其实这些语言是比较贴近人类可阅读的文法，也就是比较接近英语文法的语言。这是为了方便人类阅读及撰写，计算机其实看不懂 C/C++这类语言，为了将 C/C++翻译为 0、1 序列组成的机器指令，必须有个翻译员。担任翻译员工作的就是编译程序(Compiler)，如图 1.8 所示。

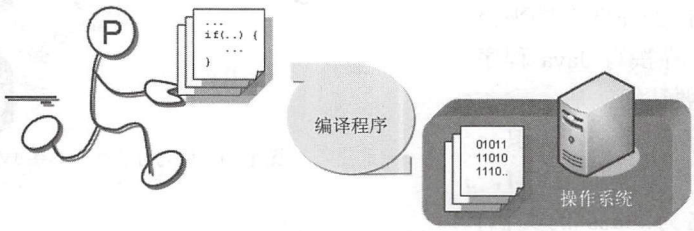


图 1.8　编译程序将程序翻译为机器码

问题在于，每个平台认识的 0、1 序列并不一样。某个指令在 Windows 上也许是 0101，在 Linux 下也许是 1010，因此必须使用不同的编译程序为不同平台编译出可执行的机器码，在 Windows 平台上编译好的程序，不能直接拿到 Linux 等其他平台执行。也就是说，应用程序无法达到“编译一次，到处执行”的跨平台目的，如图 1.9 所示。

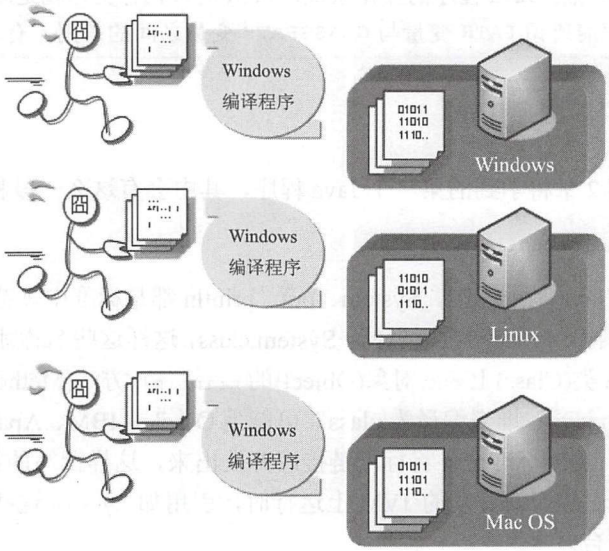


图 1.9　使用特定平台编译程序翻译出对应的机器码

Java 是个高级语言，要让计算机执行所撰写的程序，需要通过编译程序的翻译。不过用 Java 编译时，并不直接编译为相依于某平台的 0、1 序列，而是翻译为中介格式的位码(Byte Code)。

Java 原始码扩展名为.java，经过编译程序翻译为扩展名为.class 的位码。如果想要执行位码文档，目标平台必须安装 JVM(Java Virtual Machine)。JVM 会将位码翻译为相依于平台的机器码，如图 1.10 所示。

不同的平台必须安装专属该平台的 JVM。这就好比你讲中文(*.java)，Java 编译程序帮你翻译为英语(*.class)，这份英语文件到各个国家之后，再由当地看得懂英文的人(JVM)翻译为当地语言(机器码)。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

所以 JVM 担任的职责之一就是当地翻译员，将位码文档翻译为当时平台看得懂的 0、1 序列。有了 JVM，你的 Java 程序就可以达到“编译一次，到处执行”的跨平台目的。除了了解 JVM 具有让 Java 程序跨平台的重要任务之外，在撰写 Java 程序时，对 JVM 的重要认知就是：

对 Java 程序而言，只认识一种操作系统，这个系统叫 JVM，位码文档(扩展名为.class 的文档)就是 JVM 的可执行文件。

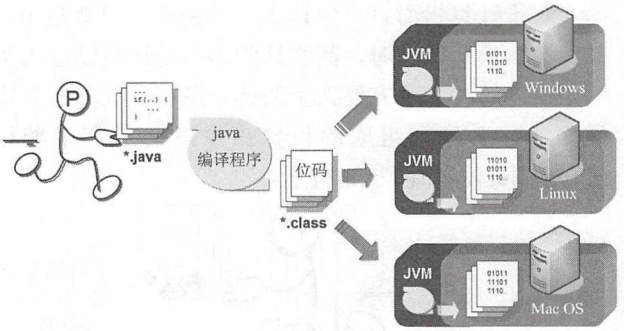


图 1.10　位码可执行于具备 JVM 的系统

Java 程序理想上并不用理会真正执行于哪个平台，只要知道如何执行于 JVM 就可以了。至于 JVM 实际上如何与底层平台沟通，则是 JVM 自己的事。由于 JVM 实际上就相当于 Java 程序的操作系统，JVM 就负责了 Java 程序的各种资源管理。

注意»» 了解“JVM 就是 Java 程序的操作系统，JVM 的可执行文件就是.class 文档”非常重要，对于以后理清所谓 PATH 变量与 CLASSPATH 变量之间的差别，有非常大的帮助。

1.2.2　区分 JRE 与 JDK

这里再看一下第 2 章将学到的第一个 Java 程序，其中会有这么一段程序代码：

```
System.out.println("Hello World");
```

前面曾经谈过，Java 是个标准，System、out、println 都是标准中规范的名称。实际上必须要有人根据标准撰写出 System.java，编译为 System.class，这样这些名称才能在撰写第一个 Java 程序时，使用 System 类(Class)上 out 对象(Object)的 println() 方法(Method)。

谁来操作 System.java？谁来编译为.class？可能是 Oracle、IBM、Apache，无论如何，这些厂商必须根据相关的 JSR 标准文件将标准链接库开发出来，这样撰写的第一个 Java 程序，在 Oracle、IBM、Apache 等厂商开发的 JVM 上运行时，引用如 System 这些标准 API，才可能轻易地运行在不同的平台。

在图 1.7 中右侧部分可以看到 Java SE API 涵盖了各式常用的链接库，像是通用的集合(Collection)、输入/输出、联机数据库的 JDBC、撰写窗口程序的 Java FX 等，这些都是在各个 JSR 标准文件规范之中。

Java Runtime Environment 就是 Java 执行环境，简称 JRE，包括 Java SE API 与 JVM。只要使用 Java SE API 中的链接库，在安装有 JRE 的计算机上就可以直接运行，无须额外在程序中再包装链接库，而可以由 JRE 直接提供，如图 1.11 所示。

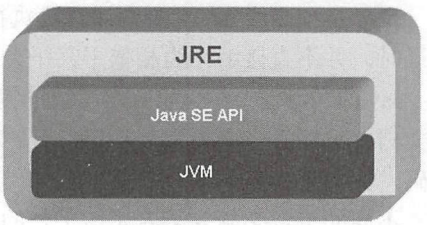


图 1.11　JRE 包括 Java SE API 与 JVM

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示>>> 在图 1.7 中还可以看到，实际上 JRE 还包括了部署(Deployment)技术，也就是如何将程序安装到客户端的技术，不过这不在本书介绍范围内。建议直接查看图 1.7 摘自的网址，其中有各种主题的文件说明，可以善加利用。

前面说过，要在.java 中撰写 Java 程序语言，使用编译程序编译为.class 文档，那么像编译程序这样的工具程序是由谁提供？答案就是 JDK，全名为 ~~Java Developer Killer~~！呃！不对！是 Java Development Kit！

正如图 1.7 所示，JDK 包括了 javac、java、javadoc 等工具程序，对于要开发 Java 程序的人，必须安装的是 JDK，这样才有这些工具程序可以使用，JDK 本身包括了 JRE，这样才能执行 Java 程序，所以总结就是“JDK 包括了 Java 程序语言、工具程序与 JRE，JRE 则包括了部署技术、Java SE API 与 JVM”，这也就是图 1.2 想表达的含义。

撰写 Java 程序才需要 JDK，如果你的程序只是想让朋友执行呢？那他只要装 JRE 就可以了，不用安装 JDK，因为他不需要 javac 这些工具程序，但他需要 Java SE API 与 JVM。

对初学者来说，JDK 确实很不友善，这大概是 Java 阵营的哲学，它会假设你懂得如何准备相关开发环境，因此装好 JDK 之后，该自己设定的变量或选项就要自己设定，JDK 不会代劳，过去戏称 JDK 全名为 Java Developer Killer 其实是有其来源的。

了解 Java SE 中 JVM、JRE 与 JDK 的关系之后，接下来应该来点实际操作了。第一步就是下载、安装 JDK。

1.2.3 下载、安装 JDK

要下载 JDK，请链接到(Java SE Downloads)以下网址：

http://www.oracle.com/technetwork/java/javase/downloads/index.html

你要下载的是 Java SE 9 中的 JDK，按照以往惯例，发布 JDK 之后，每隔几个月会针对使用者反馈的 BUG 或安全问题进行修正，并发布一个修正版 JDK。在过去，如果你连接以上网址时，出现的字样是 Java SE Development XuN 之类的字样，其中 X 会是重大特性发布用的号码，像是 8、9 等号码，也就是代表 Java SE 8、Java SE 9 等，N 就是 BUG 或安全问题更新版本号。图 1.12 所示为 JDK 的下载页面。

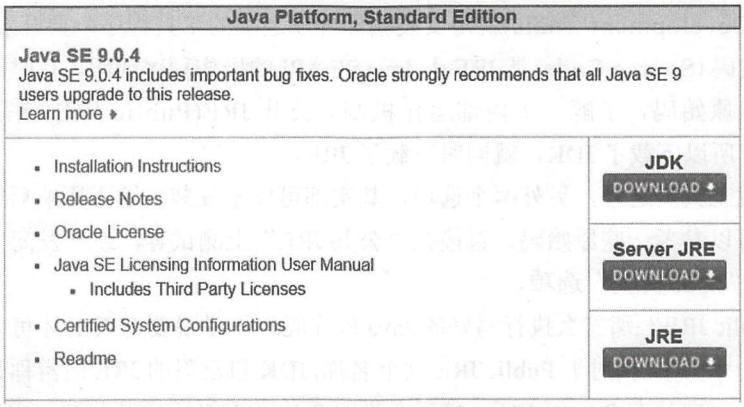


图 1.12 JDK 下载页面

然而，Oracle 在 Oracle Java SE Support Roadmap 中提出了新的特性版本发布方式，不再是基于重大特性来发布版本，而是改为基于时间、以半年为周期、持续发布小特性的版本，让一

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

些对开发者有用的小特性，可以在这些新版本中放出，而这些版本将会是短期支持版本，在下个小特性版本发布之后，上个版本就不再维护，使用者要赶快更新至新版本。

提示 »» 有些语言也是持续而频繁地发布新版本，比如 Go 语言，基于时间而非基于重大特性来发布的软件，最典型的范例就是 Ubuntu。

除了短期支持版本之外，Oracle 也有 Java SE 的长期支持版本，基本上以 3 年为周期，可以通过 Oracle Java SE Advanced、Oracle Java SE Advanced Desktop 或 Java SE Suite 取得。另外，Java SE 8 会是个长期支持版本，将支持至 2022 年，详情可以参考 *Oracle Java SE Support Roadmap* 中的说明：http://www.oracle.com/us/technologies/java/eol-135779.html

虽然未来 JDK 在小特性版本发布时，预计采用`$YEAR.$MONTH` 格式，不过看来尚有讨论空间，详情可看：mail.openjdk.java.net/pipermail/jdk-dev/2017-October/000007.html

在网页中还可以看到 Server JRE 与 JRE 的下载。前面提过，用户如果要执行 Java 程序，只需要安装 JRE，不需要安装 JDK，其中客户端应用程序可以下载 JRE 版本，如果是要运行服务器端应用程序，可以下载 Server JRE 版本。

单击 JDK 的 Download 按钮，会进入另一个页面。必须先选中 Accept License Agreement(同意许可协议)单选按钮，接着选择对应操作系统版本的 JDK。以 Windows x64 为例，单击 jdk-9_windows-x64_bin.exe 就可以下载 JDK，如图 1.13 所示。

Java SE Development Kit 9.0.4

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux	305.07 MB	jdk-9.0.4_linux-x64_bin.rpm
Linux	338.21 MB	jdk-9.0.4_linux-x64_bin.tar.gz
macOS	382.11 MB	jdk-9.0.4_osx-x64_bin.dmg
Windows	375.56 MB	jdk-9.0.4_windows-x64_bin.exe
Solaris SPARC	206.97 MB	jdk-9.0.4_solaris-sparcv9_bin.tar.gz

图 1.13　开始下载 JDK

下载完成后，双击文档就可以看到启始安装画面，直接单击“下一步”按钮看到图 1.14 所示对话框。

开发工具(Development Tools)就是安装编译程序之类的工具程序，要开发 Java 程序，就得安装它。源代码 (Source Code)是 JRE 中 Java SE API 的操作程序代码，有时候，你会需要查看 Java SE API 原始码，了解一下内部运作机制。公共 JRE(Public JRE)就是刚刚在图 1.12 中看到的 JRE，所以下载了 JDK，就同时下载了 JRE。

除了“开发工具”之外，另外两个选项，其实都可以不安装，这不影响后续的程序开发。不过为了以后可以参考一些原始码、直接在“公共 JRE”上测试等，或开发简单的数据库程序，建议选择“全部安装”选项。

不安装 Public JRE？那怎么执行写好的 Java 程序呢？不是说要有 JRE 才可以执行吗？其实 JDK 本身附有一个 JRE，相对于 Public JRE 这个名称，JDK 自己附的 JRE 通常称为 Private JRE，只要安装 JDK，一定就有 Private JRE，稍后会说明 Private JRE 的安装位置。安装 Public JRE 或自行下载 JRE 安装，会注册 Java Plugin、Web Start 等浏览器或桌面客户端必要的元件，方便需要 JRE 的应用程序使用。

严禁网络传播本PDF，违者责任自负！　　非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

在安装时注意图 1.14 中“安装到”的位置，必须记住 JDK 安装位置，之后设定 PATH 变量时会用到这个信息。单击图 1.14 中的“下一步”按钮，等待 JDK 安装完后，若在图 1.14 中选择安装“公共 JRE”，就会再出现安装 Public JRE 的画面，同样地，请记下安装位置，如图 1.15 所示。

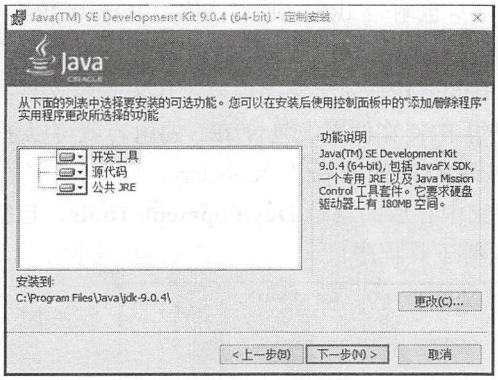


图 1.14　安装 JDK

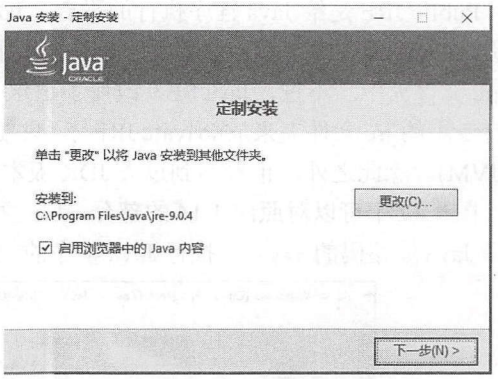


图 1.15　安装“公共 JRE”

提示 »» 基于安全原因，Oracle 曾宣布将停止浏览器外挂：
http://blogs. oracle.com/java-platform- group/ moving- to-a-plugin-free-web
就本章撰写期间，是否启用浏览器中的 Java 支持，这就交给使用者来决定，在台湾仍有些金融机构会采用 Applet 来提供网络银行之类的服务，如果你没有这类需求，在图 1.15 中可以不选中“启用浏览器中的 Java 内容”复选框。

装好 JDK 之后，在 Windows 7 或 Windows 后续版本中，可以在“开始”菜单中执行“所有程序”|“附件”|“命令提示符”命令，启动“命令提示符”窗口。在 Windows 10 中，可以直接在“开始”界面输入 cmd 启动“命令提示字符”，接着输入 java，看到图 1.16 所示界面，表示 JDK 初步安装完成。

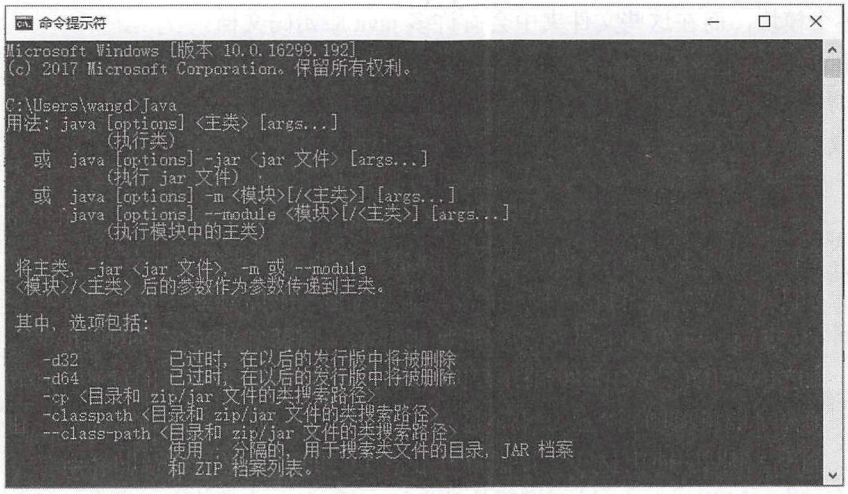


图 1.16　查看 JDK 是否安装完成

提示 »» 为什么说 JDK 初步安装完成？因为还要设定 PATH 环境变量，这要在第 2 章中再做说明。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

1.2.4 认识 JDK 安装内容

那么你到底安装了哪些东西呢？假设 JDK 与 Public JRE 各安装至 C:\Program Files\Java\jdk-9.0.4\及 C:\Program Files\Java\jre-9.0.4(如果安装时有选择“公共 JRE”选项的话)。

Public JRE 是给 Java 程序执行的平台，JDK 本身也附带 JRE，通常被称为 Private JRE。如果你熟悉 Java SE 8 或者之前版本的 Java，可能会想 Private JRE 应该还是在 JDK 安装文件夹中的 jre 文件夹中。不过，Java SE 9 改变了 JDK 与 Public JRE 的文件夹内容，JDK 安装文件夹中没有专用的 jre 文件夹来放 Private JRE 了，只要记得 JDK 文件夹中包含 JRE 的内容就可以了(以及 JVM)。除此之外，也看不到过去 JDK 版本安装后会有的 rt.jar 与 tools.jar。

图 1.17 中可以对照图 1.14 的部分，bin 文件夹中存放的就是 Development Tools，包括了编译 Java 原始码的 `javac`，执行 Java 程序的 `java` 等工具程序。

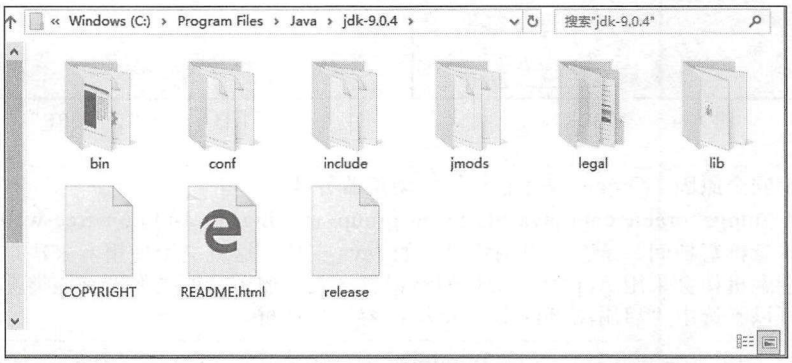


图 1.17 JDK 安装文件夹

至于 Source Code 选项，也就是 Java SE API 运行原始码的部分，可以在 lib 文件夹中的 src.zip 中找到，使用解压缩软件解开，就能看到许多文件夹，它们对应至 Java SE 9 的模块平台系统划分出来的各个模块，而在这些文件夹中会有许多.java 原始码文档。

那么 Java SE API 编译好的.class 文档放在哪里呢？Java SE 9 的模块平台系统为了改进效率、安全与维护性，使用了模块执行时期映像(Modular Run-Time Images)，又称 JIMAGE，无论是 JDK 或 Public JRE 的文件夹中，都会有个 lib 文件夹，其中有个 modules 文档，包含了.class 文档的执行时期格式。

提示>>> 可以使用 jlink 来建立专用的执行时期映像，其中只包含你使用到的模块，19.3 节会介绍这些模块如何建立；modules 文档中的.class 文档是可以取出的，如果有兴趣的话，可以查看下面的链接：http://blogs.oracle.com/sundararajan/extracting-a-single-class-file-from-ava9-platform-jimage-modules-file

至于在编译时期，Java SE 9 引入了新的 JMOD 格式来封装模块，扩展名为.jmod，这些文档位于 JDK 文件夹中的 jmods 文件夹，每个模块对应的.jmod 中就包括了编译完成的.class 文档。

提示>>> 在过去，JAR(Java Archive)文档是封装.java 或.class 文档的主要格式，有许多开发工具都能自动为用户建立 JAR 文档，而在文本模式下，还可以使用 JDK 的 jar 工具程序来制作 JAR 档案，19.3 节会介绍如何使用 jar 工具程序。

JMOD 格式可以包含比 JAR 文档更多的信息，如原生指令、链接库等，JDK9 本身包含了 jmod 工具程序，可以用来建立 JMOD 文档，或者从 JMOD 文档中取得封装的内容(目前实际上只是 ZIP 压缩，然而未来可能改变)，19.3 也会介绍如何使用 jmod 工具程序。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

你可以回头对照图 1.7，问问自己，现在是不是记住 JDK、JRE、Java SE API 与 JVM 的安装位置了呢？

注意»» 因为 Java SE 9 的 JDK/JRE 在文档实体布局上，有别于过去的 Java SE 版本，如果你曾在旧版 JDK 上使用相关开发工具撰写过 Java 程序，在升级至 Java SE 9 之后，相关开发工具可能会因此而在执行上有问题，若是如此，请确认你的开发工具是否有针对 JDK9 的升级版本。

1.3 重点复习

Java 最早是 Sun 公司“绿色项目”(Green Project)中撰写 Star7 应用程序的程序语言，当时名称不是 Java，而是取名为 Oak。全球信息网兴起，Java Applet 成为网页互动技术代表。1995 年 5 月 23 日，正式将 Oak 改名为 Java，Java Development Kits(当时的 JDK 全名)1.0a2 版本正式对外发表。

Java 2 名称从 J2SE 1.2 一直沿用至之后各个版本，直到 2006 年 12 月 11 日的 Java Platform, Standard Edition 6，改称 Java SE 6，JDK6 全名则称为 Java SE Development Kit 6，也就是不再像 Java 2 带有 2 这个号码。

2010 年，Oracle 宣布并购 Sun，Java 正式成为 Oracle 所属。

Java 根据应用领域不同，区分为 Java SE、Java EE 与 Java ME 三大平台。Java SE 是各应用平台的基础，分为四个主要的部分：JVM、JRE、JDK 与 Java 语言。JDK 包括 Java 程序语言、JRE 与开发工具，JRE 包括 Java SE API 与 JVM。

JCP 是个开放性国际组织，目的是让 Java 演进由 Sun 非正式地主导，成为全世界数以百计代表成员公开监督的过程。任何想要提议加入 Java 的功能或特性，必须以 JSR 正式文件的方式提交，经过 JCP 执行委员会投票通过，方可成为最终标准文件，有兴趣的厂商或组织可以根据 JSR 实现产品。

若 JSR 成为最终文件后，必须根据 JSR 成果做出免费且开发原始码的参考实现，称为 RI(Reference Implementation)，并提供 TCK(Technology Compatibility Kit)作为技术兼容测试工具箱，方便其他想根据 JSR 实现产品的厂商或组织参考与测试兼容性。

只有通过 TCK 兼容性测试的实现，才可以使用 Java 这个商标。

在 2006 年的 JavaOne 大会上，Sun 宣告对 Java 开放源代码，从 JDK7 b10 开始有了 OpenJDK，并于 2009 年 4 月 15 日正式发布 OpenJDK。Oracle 时代发布的 JDK7 正式版本，指定了 OpenJDK7 为官方参考实现。

OpenJDK6 并不是 Sun JDK6 的分支，而是将 OpenJDK7 中 JDK7 的特性删掉，使之符合 JDK6 的规范，因而 OpenJDK6 实际上是 OpenJDK7 的分支。

Java 编译时并不直接编译为相依于某平台的 0、1 序列，而是翻译为中介格式的位码(Byte Code)。对 Java 程序而言，只认识一种操作系统，这个系统叫 JVM，位码文档(扩展名为.class 的文档)就是 JVM 的可执行文件。

Java Runtime Environment 就是 Java 执行环境，简称 JRE，包括 Java SE API 与 JVM。只要使用 Java SE API 中的链接库，在安装有 JRE 的计算机上就可以直接运行，无须额外在程序中再包装链接库，而由 JRE 直接提供。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

JDK本身附有一个JRE，相对于Public JRE这个名称，JDK所附的JRE通常称为Private JRE，只要安装JDK，一定就有Private JRE。安装Public JRE或自行下载JRE安装，会注册Java Plugin、Web Start等浏览器或桌面客户端必要的组件，方便需要JRE的桌面应用程序使用。

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 2

从 JDK 到 IDE

学习目标

- 了解与设定 PATH
- 了解与指定 CLASSPATH、SOURCEPATH
- 使用 package 与 import 管理类别
- 初识模块平台系统
- 初识 JDK 与 IDE 的对应关系

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

2.1 从 Hello World 开始

第一个 Hello World 的出现是在 Brian Kernighan 写的 *A Tutorial Introduction to the Language B* 一书中(B 语言是 C 语言的前身)，用来将 Hello World 文字显示在计算机屏幕上，自此之后，很多的程序语言教学文件或书籍上，已经无数次地将它当作第一个范例程序。为什么要用 Hello World 来当作第一个程序范例？因为它很简单，初学者只要输入简单几行程序(甚至一行)，即可要求计算机执行指令并得到反馈：显示 Hello World。

本书也要从显示 Hello World 开始，然而，在完成这个简单的程序之后，千万要记得，探索这个简单程序之后的种种细节。千万别过于乐观地以为，你想从事的程序设计工作就是如此容易驾驭。

2.1.1 撰写 Java 原始码

在正式撰写程序之前，请先确定你可以看到文档的扩展名。在 Windows 下默认不显示扩展名，这会造成重新命名文档时的困扰，如果目前在“资源管理器”下无法看到扩展名，在 Windows 7 下执行“组织”|“文件夹和搜索选项”命令，或在 Windows 8 及之后的 Windows 版本下执行“查看”|“选项”命令，打开“文件夹选项”对话框后都是切换至“查看”选项卡，取消选择“隐藏已知文件类型的扩展名”复选框，如图 2.1 所示。

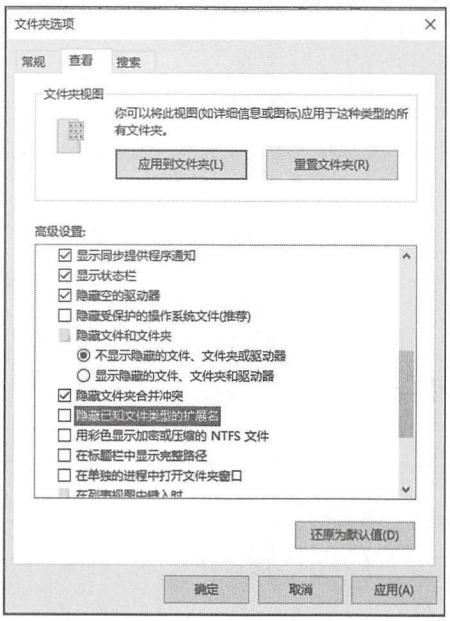


图 2.1 取消选择“隐藏已知文件类型的扩展名”复选框

接着选择一个文件夹来撰写 Java 原始码文档。本书都是在 C:\workspace 文件夹中撰写程序，请新创建一个“文本文件”(也就是.txt 文件)，并将其命名文件为 HelloWorld.java。由于将文字文件的扩展名从.txt 改为.java，系统会询问是否更改扩展名，请确定更改，接着右击 HelloWorld.java，从弹出的快捷菜单中选择“编辑”命令，并撰写程序，如图 2.2 所示。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

图 2.2 第一个 Java 程序

提示 »» 在 Windows 中内建的记事本编辑器并不是很好用，建议使用 NotePad++：http://notepad-plus-plus.org/

这个文档撰写时有几点必须注意：

- 扩展名是 .java。这也就是你必须让“资源管理器”显示扩展名的原因。
- 主文档名与类名称必须相同。类名称是指 `class` 关键词(Keyword)后的名称，这个范例就是 `HelloWorld` 这个名称，这个名称必须与 HelloWorld.java 的主文档名(HelloWorld)相同。
- 注意每个字母的大小写。Java 程序区分字母大小写，`System` 与 `system` 对 Java 程序来说是不同的名称。
- 空格只能是半角空格符或 Tab 字符。有些初学者可能不小心输入了全角空格符，这很不容易检查出来。

老实说，要对新手解释第一个 Java 程序并不容易，这个简单的程序就涉及文档管理、类(`Class`)定义、程序进入点、命令行自变量(Command Line Argument)等概念。以下先针对这个范例做基本说明。

1. 定义类

`class` 是用来定义类的关键词，之后接上类名称(`HelloWorld`)。Java 程序规定，所有程序代码都要定义在“类”中。`class` 前有个 `public` 关键词，表示 `HelloWorld` 类是公开类。就目前为止你只要知道，一个.java 文档可定义多个类，但是只能有一个公开类，而且主文档名必须与公开类名称相同。

2. 定义区块(Block)

在程序中使用大括号“{”与“}”定义区块，大括号两两成对，目的在于区别程序代码范围。例如，程序中 `HelloWorld` 类的区块包括了 `main()` 方法(Method)，而 `main()` 方法的区块包括了一句显示信息的程序代码。

3. 定义 main() 方法

程序执行的起点就是程序进入点(Entry Point)，Java 程序执行的起点是 `main()` 方法。规格书中规定 `main()` 方法的形式一定得是：

```
public static void main(String[] args)
```

提示 »» 虽然说是书中的规定，不过其实日后你理解每个关键词的意义，还是可以就每个元素加以解释。`main()` 方法是 `public` 成员，表示可以被 JVM 公开执行；`static` 表示 JVM 不用生成类实例就可以调用；Java 程序执行过程的错误，都是以例外方式处理，所以 `main()` 不用传回值，声明为 `void` 即可；`String[] args` 可以在执行程序时，取得用户指定的命令行自变量。

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

4. 撰写描述(Statement)

来看 main()中的一行描述：

```
System.out.println("Hello World");
```

描述是程序语言中的一行指令，简单地说，就是程序语言中的“一句话”。注意每句描述的结束要用分号(;)，这句描述的作用，就是请系统的输出装置显示一行文字 Hello World。

提示»» 其实你使用了 java.lang 包(package)中 System 类的 public static 成员 out，out 参考至 PrintStream 实例，你使用 PrintStream 定义的 println()方法，将指定的字符串(String)输出至文本模式上。println()表示输出字符串后换行，如果使用 print()，输出字符串后不会换行。

其实我真正想说的是：一个基本的 Java 程序这么写就对了。一下子要接受如此多概念确实不容易，如果现阶段无法了解，就先当这些是 Java 语法规范，相关元素在本书之后各章节还会详细解释，届时自然就会了解第一个 Java 程序是怎么一回事了。

2.1.2 PATH 是什么

在第 1 章介绍过，*.java 必须编译为*.class 才可以在 JVM 中执行，Java 的编译工具程序是 javac。装好 JDK 之后，工具程序就会放在 JDK 安装文件夹的 bin 文件夹中，你必须按照第 1 章介绍的方法打开“命令提示符”模式，切换至 C:\workspace，并执行 javac 指令，如图 2.3 所示。

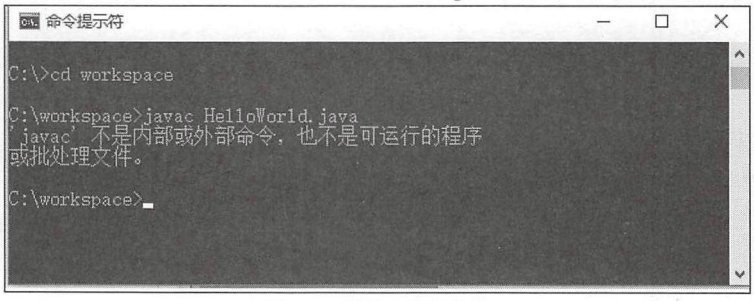


图 2.3 喔喔！执行失败

失败了？为什么？这是操作系统 Windows 在跟你抱怨：它找不到 javac 放在哪里！当要执行一个工具程序，那个指令放在哪里，系统默认是不知道的，除非你告诉系统工具程序存放的位置，如图 2.4 所示。

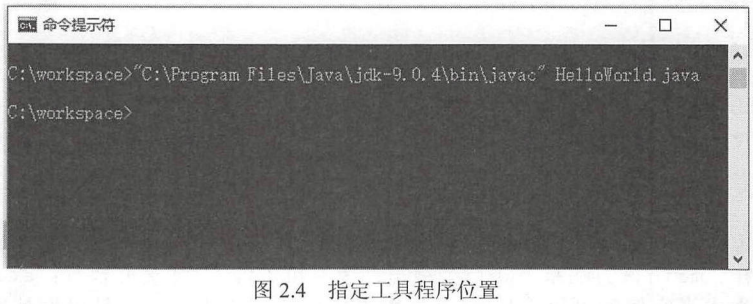


图 2.4 指定工具程序位置

javac 编译成功后会静悄悄地结束，所以没看到信息就是好消息，但是这样下指令实在太麻烦了，而且你会有疑问：第 1 章安装 JDK 最后示范执行 java 指令时，为什么不用指定位置？

非卖品，仅供非商业用途或交流学习使用

当你输入一个指令而没有指定路径信息时，操作系统会依照 PATH 环境变量中设定的路径顺序，依次寻找各路径下是否有这个指令。可以执行 echo %PATH%来看看目前系统 PATH 环境变量中包括哪些路径信息，如图 2.5 所示。

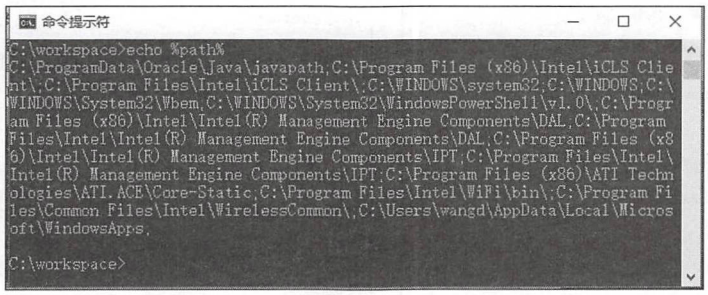


图 2.5　查看 PATH 信息

根据图 2.5 中的 PATH 信息，如果输入 java 指令，系统会从第一个路径开始找有无 java(.exe)工具程序，如果没有，再找下一个路径有无 java(.exe)工具程序……找到的话就执行。

如果你安装了 Java SE 9 的“公共 JRE”，安装程序会自动在 PATH 中加入 C:\ProgramData\Oracle\Java\javapath，查看该路径的话，你会发现其中就包括了 java.exe 工具程序，这就是为何第 1 章安装 JDK(JRE)后，就可以直接执行 java 指令的原因。

提示 »» 若是旧版 JDK，在安装完成后，会自动放一份 java(.exe)到 C:\Windows\system32。

然而按照图 2.5 中的 PATH 信息，如果输入 javac 指令，系统即使找完 PATH 中所有路径后，都不会找到 javac(.exe)工具程序，当所有路径都找不到指定的工具程序时，就会出现图 2.3 所示的错误信息。

你要在 PATH 中设定工具程序的路径信息，系统才可以在 PATH 中找到你要执行的指令。如果要设定 PATH，Windows 中可以使用 SET 指令来设定，设定方式为 SET PATH=路径，如图 2.6 所示。

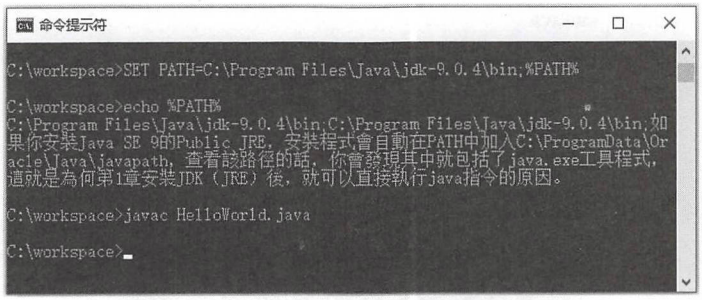


图 2.6　设定 PATH 环境变量

设定时若有多个路径，会使用分号(;)作分隔，通常会将原有 PATH 附加在设定值后面，这样寻找其他指令时，才可以利用原有的 PATH 信息。设定完成之后，就可以执行 javac 而不用额外指定路径。

不过在“命令提示符”模式中设定，关掉这个“命令提示符”模式后，下次要开启“命令提示符”模式又要重新设定。为了方便，可以在“用户环境变量”或“系统环境变量”中设定 PATH。在 Windows 7 之后的 Windows 版本中可以右击“此电脑”图标，在弹出的快捷菜单中选择“属性”命令，在打开的窗口中单击“高级系统设置”超链接，进入“系统属性”对话框，

接着切换至“高级”选项卡，单击“环境变量”按钮，在“环境变量”对话框的“wangd 的用户变量”或“系统变量”列表中编辑 PATH 变量，如图 2.7 所示。

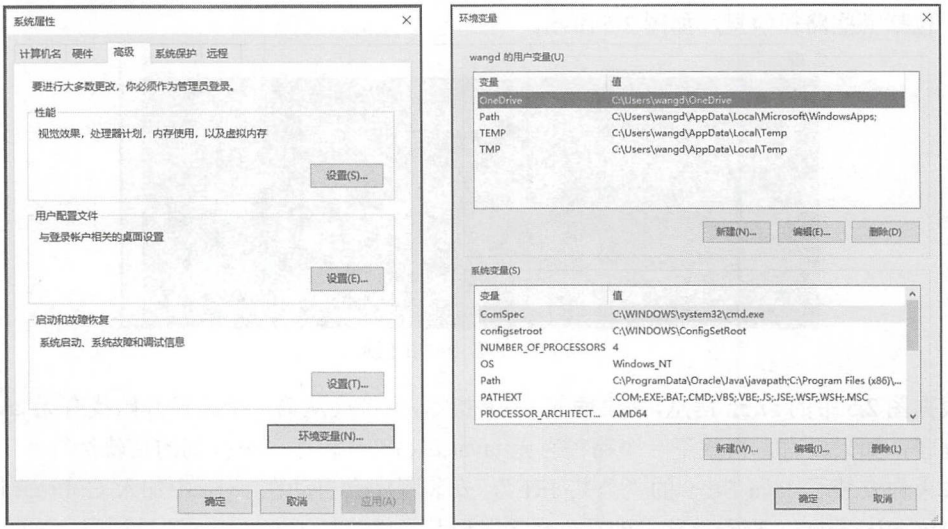


图 2.7　设定用户变量或系统变量

在一个可以允许多人共享的系统中，系统环境变量的设定会套用至每个登录的用户，而用户环境变量只影响个别用户。开启一个“命令提示符”模式时，获得的环境变量会是系统环境变量再“附加”用户环境变量。如果使用 SET 指令设定环境变量，则以 SET 设定的结果决定。

以 Windows 10 设定系统变量为例，可在图 2.7 中选取 PATH 变量后，单击“编辑”按钮，打开“编辑环境变量”对话框，然后单击“新建”按钮，在变量列表中输入 JDK 的 bin 目录的路径(C:\Program Files\Java\jdk-9.0.4\bin)，然后单击“上移”按钮，将设定值放置在 Path 的最上面，接着单击“确定”按钮完成设定，如图 2.8 所示。重新启动“命令提示符”之后，就会套用新的环境变量。

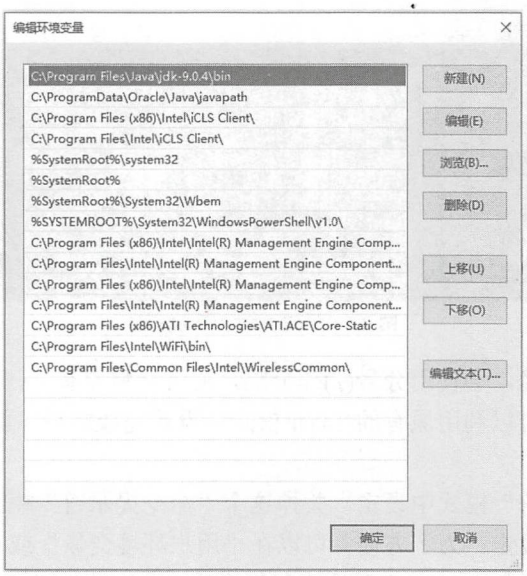


图 2.8　编辑 Path 系统变量

提示 »» 若是旧版本 Windows，单击“编辑”按钮，在“变量值”字段的最前面输入 JDK 的 bin 目录的路径(C:\Program Files\Java\jdk-9.0.4\bin)，然后紧跟着一个分号，作为路径设定区隔，接着单击“确定”按钮完成设定，同样必须重新开启命令提示字符，才会套用新的环境变量。

建议将 JDK 的 bin 路径放在 PATH 变量的最前方，是因为系统搜索 PATH 路径时，会从最前方开始，如果在路径下找到指定的工具程序就会直接执行。当系统中安装两个以上 JDK 时，PATH 路径中设定的顺序，将决定执行哪个 JDK 下的工具程序。在安装了多个 JDK 或 JRE 的计算机中，确定执行了哪个版本的 JDK 或 JRE 非常重要，确定 PATH 信息是一定要做的动作。

提示 »» 由于打开“命令提示符”模式时获得的环境变量会是系统环境变量附加用户环境变量，若系统环境变量 PATH 中已经设定好某个 JDK，即使你在用户环境变量 PATH 中将想要的 JDK 路径设在最前头，也会执行到系统环境变量 PATH 中设定的 JDK。如果你有足够权限修改系统环境变量，建议修改系统环境变量 PATH。如果没有权限变更，那就使用 SET 指令，因为使用 SET 指令设定环境变量，会以 SET 设定的结果决定。

2.1.3 JVM(java)与 CLASSPATH

在图 2.6 中完成编译 HelloWorld.java 之后，相同文件夹下就会出现 HelloWorld.class。在第 1 章说过，JVM 的可执行文件扩展名是.class，接下来要启动 JVM，要求 JVM 执行 HelloWorld，如图 2.9 所示。

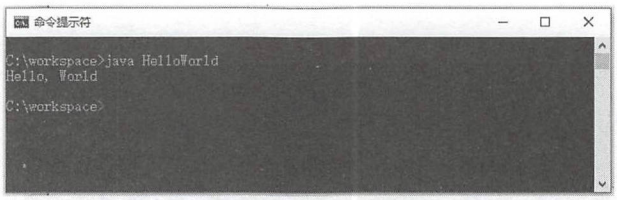


图 2.9 第一个 Hello，World 出现了

在图 2.9 中，启动 JVM 的指令是 java，而要求 JVM 执行 HelloWorld 时，只要指定类名称(就像执行 javac.exe 工具程序，只要输入 javac 就可以了)，不用附加.class 扩展名，附加.class 反而会有错误信息。

注意 »» Java SE 9 加入了模块平台系统此一重大特性，就结论而言，模块平台系统建议未来使用模块路径(Module path)来取代类路径(Class path)。然而过去 20 多年来，许多 Java 项目是基于类路径，模块路径要取代类路径并非一朝一夕之事，因而在实践方面，你还是要了解接下来介绍的类路径，有此基础之后，在学习模块平台系统时，才能掌握模块路径的运用。

接下来，请切换至 C:\，想想看，如何执行 HelloWorld？以下几个方式都是行不通的，如图 2.10 所示。

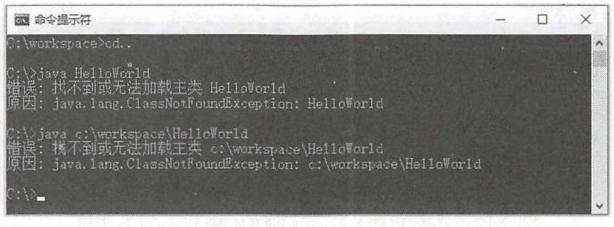


图 2.10 怎么执行 HelloWorld?

你要知道 java 这个指令是做什么用的？正如前面所讲，执行 java 指令的目的在于启动 JVM，之后接着类名称，表示要求 JVM 执行指定的可执行文件(.class)。

在 2.1.2 节中提过，实体操作系统下执行某个指令时，会根据 PATH 中的路径信息，试图找到可执行文件(例如对 Windows 来说，就是.exe、.bat 扩展名的文档，对 Linux 等就是有执行权限的文档)。

从第 1 章就一直强调，JVM 是 Java 程序唯一识别的操作系统，对 JVM 来说，可执行文件就是扩展名为.class 的文档。想在 JVM 中执行某个可执行文件(.class)，就要告诉 JVM 这个虚拟操作系统到哪些路径下寻找文档，方式是通过 CLASSPATH 指定其可执行文件(.class)的路径信息。

用 Windows 与 JVM 做个简单的对照，就可以很清楚地对照 PATH 与 CLASSPATH，如表 2.1 所示。

表 2.1 PATH 与 CLASSPATH

操作系统	搜索路径	可执行文件
Windows	PATH	.exe、.bat
JVM	CLASSPATH	.class

提示 »» PATH 与 CLASSPATH 根本就是不同层次的环境变量，实际操作系统搜索可执行文件是看 PATH，JVM 搜索可执行文件(.class)只看 CLASSPATH。

如何在启动 JVM 时告知可执行文件(.class)的位置？可以使用-classpath 自变量来指定，如图 2.11 所示。

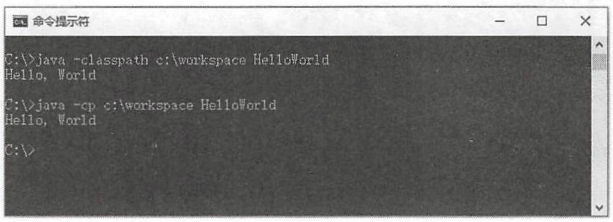


图 2.11 启动 JVM 时指定 CLASSPATH

如图 2.11 中，-classpath 有个缩写形式-cp，这比较常用。如果有多个路径信息，则可以用分号分隔。例如：

```
java -cp C:\workspace;C:\classes HelloWorld
```

JVM 会按 CLASSPATH 路径顺序，搜索是否有对应的类文档，先找到先载入。如果在 JVM 的 CLASSPATH 路径信息中都找不到指定的类文档，JDK7 前的版本会出现 java.lang.NoClassDefFoundError 信息，而 JDK7 之后的版本会有比较友善的中文错误信息，如图 2.10 所示。

为什么图 2.9 不用特别指定 CLASSPATH 呢？JVM 默认的 CLASSPATH 就是读取目前文件夹中的.class，如果自行指定 CLASSPATH，则以指定的为主，如图 2.12 所示。

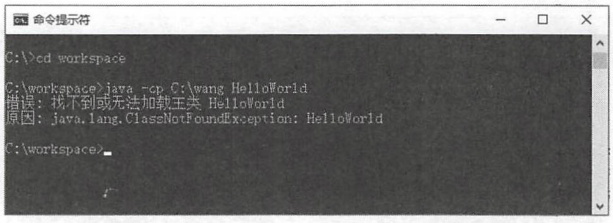


图 2.12 在指定的 CLASSPATH 中找不到类文档

在图 2.12 中，虽然工作路径是在 C:\workspace(其中有 HelloWorld.class)，你启动 JVM 时指定到 C:\xyz 中搜索类文档，JVM 还是老实地到指定的 C:\wang 中找寻，结果当然就是找不到而显示错误信息。如果希望从当前文件夹开始寻找类文档，则可以使用“.”指定，如图 2.13 所示。

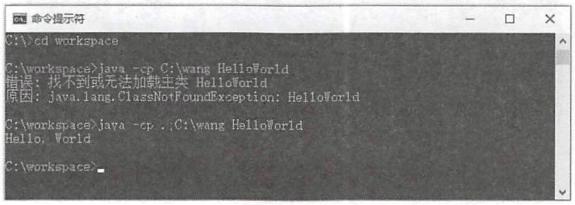


图 2.13　指定“.”表示搜索类文档时包括目前文件夹

如果使用 Java 开发了链接库，这些链接库中的类文档会封装为 JAR(Java Archive)文档，也就是扩展名为.jar 的文档。JAR 文档实际使用 ZIP 格式压缩，当中包含一些.class 文档。那么，如果你有个 JAR 文档，如何在 CLASSPATH 中设定？

答案是将 JAR 文档当作特别的文件夹，例如，有 abc.jar 与 xyz.jar 文档放在 C:\lib 下，执行时若要使用 JAR 文档中的类文档，可以如下：

```
java -cp C:\workspace;C:\lib\abc.jar;C:\lib\xyz.jar SomeApp
```

如果有些类路径经常使用，其实也可以通过环境变量设定。例如：

```
SET CLASSPATH=C:\classes;C:\lib\abc.jar;C:\lib\xyz.jar
```

在启动 JVM 时，也就是执行 java 时，若没使用-cp 或-classpath 指定 CLASSPATH，就会读取 CLASSPATH 环境变量。同样地，“命令提示符”模式中的设定在关闭该模式之后就会失效。如果希望每次开启“命令提示符”模式都可以套用某个 CLASSPATH，也可以设定在系统变量或用户变量中。如果执行时使用了-cp 或-classpath 指定 CLASSPATH，则以-cp 或-classpath 的指定为主。

如果某个文件夹中有许多.jar 文档，从 Java SE 6 开始，可以使用“*”表示使用文件夹中所有.jar 文档。例如，指定使用 C:\jars 下所有 JAR 文档：

```
java -cp .;C:\jars\* cc.openhome.JNotePad
```

在 Java SE 6 中，CLASSPATH 新的指定方式也适用在系统环境变量的设定上。

提示 »» 可以使用 JDK 内置的 jar 来创建 JAR 文档，不过很少会这么做，因为开发时会有相关工具程序来协助创建。如果对于 jar 的基本使用有兴趣，可以参考 19.3 节。

2.1.4　编译程序(javac)与 CLASSPATH

在范例文件 labs/CH02 文件夹中有个 classes 文件夹，请将其复制至 C:\workspace 中，确认 C:\workspace\classes 中有个已编译的 Console.class 文件。可以在 C:\workspace 中建个 Main.java 文档，使用 Console 类，如图 2.14 所示。

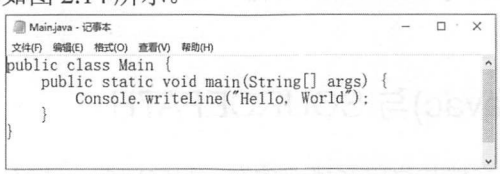


图 2.14　使用已编译好的.class 文档

如果按照图 2.15 所示编译，将会出现错误信息。

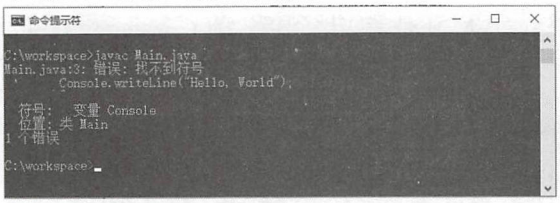


图 2.15　找不到 Console 类的编译错误

编译程序在抱怨，它找不到 Console 类在哪里(找不到符号)。事实上，在使用 javac 编译程序时，如果要使用到其他类链接库，也必须指定 CLASSPATH，告诉 javac 编译程序到哪里寻找.class 文档，如图 2.16 所示。

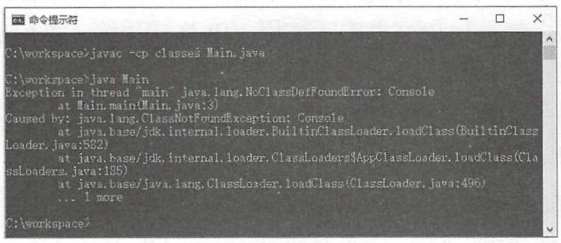


图 2.16　编译成功，但执行时找不到 Console 类的错误

这一次编译成功了，但无法执行，原因是执行时找不到 Console 类。因为你执行时忘了为 JVM 指定 CLASSPATH，所以 JVM 找不到 Console 类。如果按照图 2.17 执行就可以了。

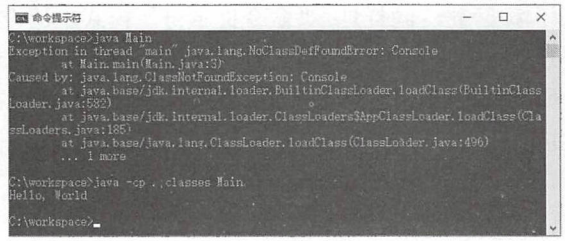


图 2.17　找到 Console 与 Main 执行成功

别忘了，如果执行 JVM 时指定了 CLASSPATH，就只会在指定的 CLASSPATH 中寻找使用到的类，所以图 2.17 指定 CLASSPATH 时，是指定.;classes，注意一开始的“.”，这表示当前文件夹，这样才可以找到当前文件夹下的 Main.class 及 classes 下的 Console.class。

2.2　管理原始码与位码文档

来观察一下 C:\workspace，原始码文档(.java)与位码文档(.class)都放在一起，想象一下，如果程序规模稍大，一堆.java 与.class 文档还放在一起，会有多么混乱，你需要有效率地管理原始码与位码文档。

2.2.1　编译程序(javac)与 SOURCEPATH

首先必须解决原始码文档与位码文档都放在一起的问题。请将范例文件中 labs 文件夹的

Hello1 文件夹复制至 C:\workspace 中，Hello1 文件夹中有 src 与 classes 文件夹，src 文件夹中有 Console.java 与 Main.java 两个文档，其中 Console.java 就是 2.1.4 节中 Console 类的原始码(目前你不用关心它如何撰写)，而 Main.java 的内容与图 2.14 所示相同。

简单地说，src 文件夹将用来放置原始码文档，而编译好的位码文档希望能指定存放至 classes 文件夹。可以在“命令提示符”模式下，切换至 Hello1 文件夹，然后进行编译，如图 2.18 所示。

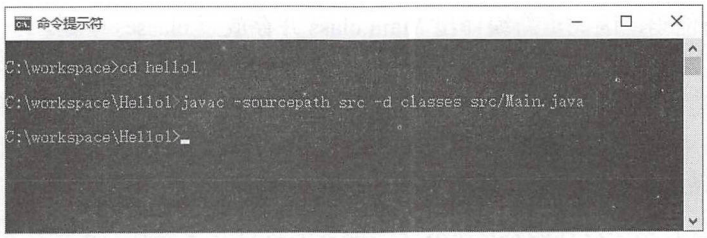


图 2.18 指定-sourcepath 与-d 进行编译

在编译 src/Main.java 时，由于程序代码中要使用到 Console 类，你必须告诉编译程序，Console 类的原始码文档存放位置。这里使用-sourcepath 指定从 src 文件夹中寻找原始码文档，而-d 指定了编译完成的位码存放文件夹，编译程序会将使用到的相关类原始码也一并进行编译，编译完成后，会在 classes 文件夹中看到 Console.class 与 Main.class 文档。执行图 2.19 所示的程序。

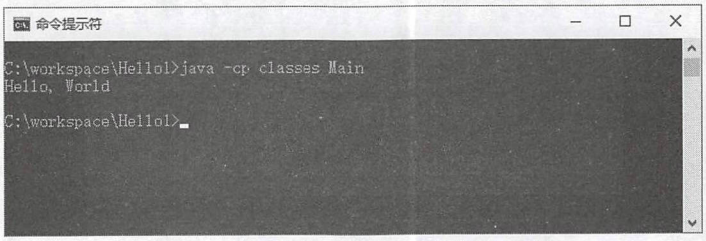


图 2.19 指定执行 classes 中的 Main 类

可以在编译时指定-verbose 自变量，看到编译程序进行编译时的过程，有助于了解 SOURCEPATH 与 CLASSPATH 的差别，如图 2.20 所示。

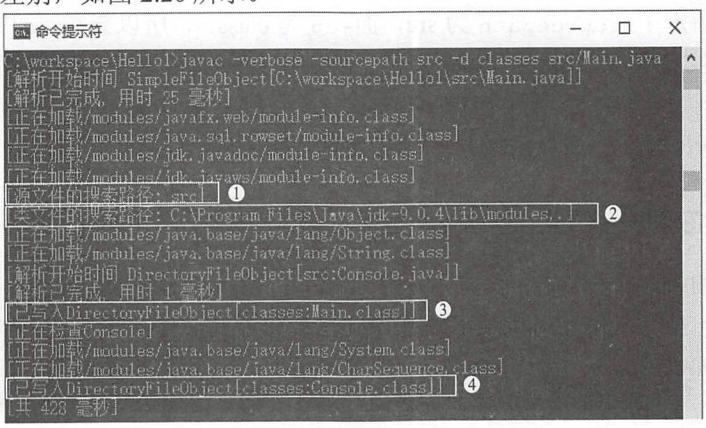


图 2.20 编译时指定-verbose

就初学者而言，最主要看看圈住的部分。在编译时，会先搜索-sourcepath 指定的文件夹

(上例指定 src)❶是不是使用到了类原始码，然后搜索是否有已编译的类位码❷。你可以发现，默认搜寻位码的路径包括了 JDK 文件夹中的 lib\modules，以及目前的工作路径，也就是最后那个“.”。

确认原始码与位码搜索路径之后，接着检查类位码的搜寻路径中，是否已经有编译完成的 `Main` 类。如果存在且从上次编译后，`Main` 类的原始码并没有改变，则无须重新编译；如果不存在，则重新编译 `Main` 类。就上例而言，由于类位码的搜寻路径中并不包括 classes 文件夹，所以找不到 `Main` 类位码，因此重新编译出 Main.class 并存放至 classes 中❸。

接着检查类位码的搜寻路径中，是否已经有编译完成的 `Console` 类，如果存在且从上次编译后，`Console` 类的原始码并没有改变，则无须重新编译；如果不存在，则重新编译 `Console` 类。就上例而言，由于类位码的搜寻路径中并不包括 classes 文件夹，所以找不到 `Console` 类位码，因此重新编译出 Console.class 并存放至 classes 中❹。

实际项目中会有数以万计的类，如果每次都要重新将.java 编译为.class，那会是非常费时的工作，所以编译时若类路径中已存在位码，且上次编译后，原始码并没有修改，无须重新编译，会比较节省时间。因此，就上例而言，可以指定`-cp` 为 classes，如图 2.21 所示。

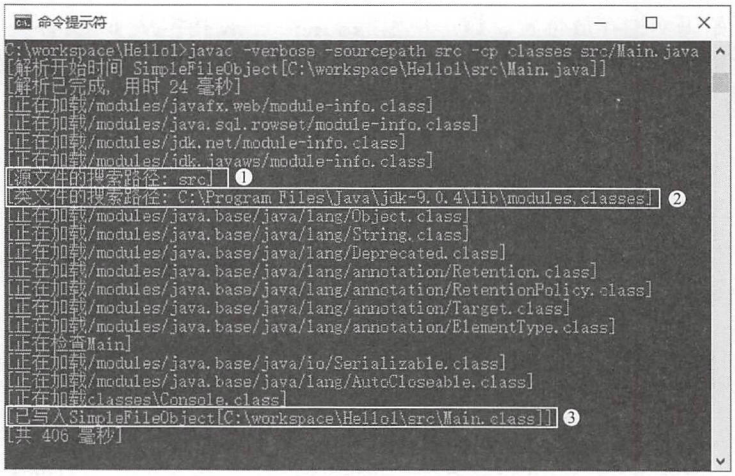


图 2.21　编译时指定`-sourcepath` 与`-cp`

注意，这次指定了`-sourcepath` 为 src，而`-cp` 为 classes，所以会在 src 中搜索位原始码文档❶，除了 JDK 文件夹中的 lib\modules 文档之外，也会在 classes 中搜索位码文档❷(注意最后的 classes)。由于类位码的搜寻路径中包括 classes 文件夹，所以找到 `Console` 类位码，因此无须重新编译 Console.class，而只编译 `javac` 指定的 Main.java 为 Main.class❸。

2.2.2　使用 package 管理类

现在所撰写的类，.java 放在 src 文件夹中，编译出来的.class 放置在 classes 文件夹下，就文档管理上比较好一些了，但还不是很好，就如同你会分不同文件夹来放置不同作用的文档，类也应该分门别类加以放置。

举例来说，一个应用程序中会有多个类彼此合作，也有可能由多个团队共同分工，完成应用程序的某些功能块，再组合在一起。如果你的应用程序是多个团队共同合作，若不分门别类放置.class，那么若 A 部门写了个 `Util` 类并编译为 Util.class，B 部门写了个 `Util` 类并编译为

Util.class，当他们要将应用程序整合时，就会发生文档覆盖的问题，而如果现在要统一管理原始码，也许原始码也会发生彼此覆盖的问题。

你要有个分门别类管理类的方式，无论实体文档上的分类管理，还是类名称上的分类管理，在 Java 语法中，有个 **package** 关键词，可以协助你达到这个目的。

Lab 请用“记事本”打开 2.2.2 节中 Hello1/src 文件夹中的 Console.java，在开头输入图 2.22 所示的反白文字。

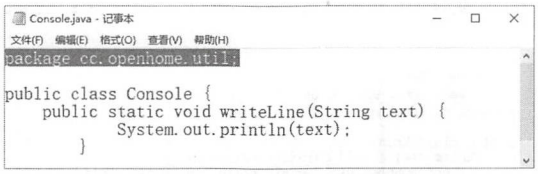


图 2.22 将 Console 类放在 cc.openhome.util 类下

这表示，Console 类将放在 cc.openhome.util 类下，用 Java 的术语来说，Console 这个类将放在 cc.openhome.util 包(package)。

再用“记事本”打开 2.2.2 节中 Hello1/src 文件夹中的 Main.java，在开头输入图 2.23 所示的反白文字，这表示 Main 类将放在 cc.openhome 类中。

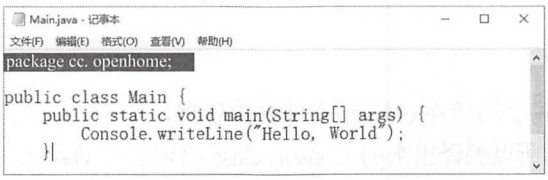


图 2.23 将 Main 类放在 cc.openhome 类下

提示 »» 包通常会用组织或单位的网址命名。举例来说，我的网址是 openhome.cc，包就会反过来命名为 cc.openhome。由于组织或单位的网址是独一无二的，这样的命名方式，不会与其他组织或单位的包名称发生同名冲突。

当类原始码开始使用 package 进行分类时，会具有四种管理上的意义：

- 原始码文档要放置在与 package 所定义名称层级相同的文件夹层级中。
- package 所定义名称与 class 所定义名称，会结合而成类的完全吻合名称(Fully Qualified Name)。
- 位码文档要放置在与 package 所定义名称层级相同的文件夹层级中。
- 要在包间可以直接使用的类或方法(Method)必须声明为 public。

关于第四点，涉及包间的权限管理，将在 5.2.1 节介绍，本章先不予讨论，以下针对前三点分别做说明。

1. 原始码文档与包管理

目前计划将所有原始码文档放在 src 文件夹中管理，由于 Console 类使用 package 定义在 cc.openhome.util 包下，所以 Console.java 必须放在 src 文件夹中的 cc/openhome/util 文件夹，在没有工具辅助下，必须手动建立文件夹。Main 类使用 package 定义在 cc.openhome 包下，所以 Main.java 必须放在 src 文件夹的 cc/openhome 文件夹中。

这么做的好处很明显，日后若不同组织或单位的原始码要放置在一起管理，就不容易发生原始码文档彼此覆盖的情况。

2. 完全吻合名称

由于 Main 类是位于 cc.openhome 包分类中，其完全吻合名称是 cc.openhome.Main，而 Console 类位于 cc.openhome.util 分类中，其完全吻合名称为 cc.openhome.util.Console。

在原始码中指定使用某个类时，如果是相同包中的类，只要使用 class 所定义的名称即可，而不同包的类，必须使用完全吻合名称。由于 Main 与 Console 类是位于不同的包中，在 Main 类中要使用 Console 类，就必须使用 cc.openhome.util.Console。也就是说，Main.java 现在必须修改，如图 2.24 所示。

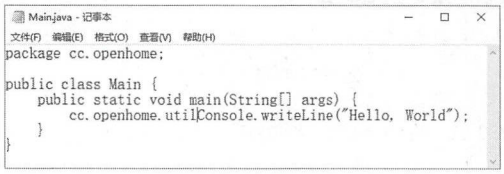


图 2.24 使用完全吻合名称

这么做的好处在于，若另一个组织或单位也使用 class 定义了 Console，但其包定义为 com.abc，则其完全吻合名称为 com.abc.Console，也就不会与 cc.openhome.util.Console 发生名称冲突问题。

3. 位码文档与包管理

目前计划将所有位码文档放在 classes 文件夹中管理，由于 Console 类使用 package 定义在 cc.openhome.util 包下，所以编译出来的 Console.class 必须放在 classes 文件夹的 cc/openhome/util 文件夹中。Main 类使用 package 定义在 cc.openhome 包下，所以 Main.class 必须放在 classes 文件夹的 cc/openhome 文件夹中。

不用手动建立对应包层级的文件夹，在编译时如使用-d 指定位码的存放位置，就会自动建立对应包层级的文件夹，并将编译出来的位码文档放置至应有的位置，如图 2.25 所示。

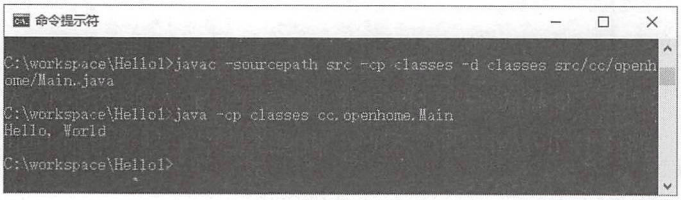


图 2.25 指定-d 自变量，会建立对应包的文件夹层级

注意，由于 Main 类位于 cc.openhome 包中，所以图 2.25 使用 java 执行程序时，必须指定完全吻合名称，也就是指定 cc.openhome.Main 这个名称。

2.2.3 使用 import 偷懒

使用包管理，解决了实体文档与撰写程序时类名称冲突的问题，但若每次撰写程序时，都得输入完全吻合名称，也是件麻烦的事。想想看，有些包定义的名称很长时，单是要输入完全吻合名称得花多少时间。

可以用 import 偷懒一下，如图 2.26 所示。

编译与执行时的指令方式与图 2.25 相同。当编译程序剖析 Main.java 看到 import 声明时，

会先记得 import 的名称，后续剖析程序时，若看到 Console 名称，原本会不知道 Console 是什么东西，但编译程序记得你用 import 告诉过它，如果遇到不认识的名称，可以比对一下 import 过的名称，编译程序试着使用 cc.openhome.util.Console，结果可以在指定的类路径(cc/openhome/util 文件夹)下找到 Console.class，并进行编译。

图 2.26 使用 import 减少打字麻烦

所以 import 只是告诉编译程序，遇到不认识的类名称，可以尝试使用 import 过的名称，import 让你少打一些字，让编译程序多为你做一些事。

如果同一包下使用到多个类，便会多次使用 import：

```
import cc.openhome.Message;
import cc.openhome.User;
import cc.openhome.Address;
```

可以更偷懒一些，用以下的 import 语句：

```
import cc.openhome.*;
```

图 2.26 也可以使用以下的 import 语句，其编译与执行结果相同：

```
import cc.openhome.util.*;
```

当编译程序剖析 Main.java 看到 import 的声明时，会先记得有 cc.openhome.util 包名称，在后续剖析到 Console 名称时，发现它不认识 Console 是什么东西，但编译程序记得你用 import 告诉过它。若遇到不认识的名称，可以比对一下 import 过的名称，编译程序试着将 cc.openhome.util 与 Console 结合为 cc.openhome.util.Console，结果可以在指定的类路径中，cc/openhome/util 文件夹下找到 Console.class，并进行编译。

偷懒也是有个限度，如果自己写了一个 Arrays：

```
package cc.openhome;
public class Arrays {
...
}
```

若在某个类中撰写了以下的程序代码：

```
import cc.openhome.*;
import java.util.*;
public class Some {
    public static void main(String[] args) {

        Arrays arrays;
          ...
    }
}
```

那么在编译时，会发现有图 2.27 所示的错误信息。

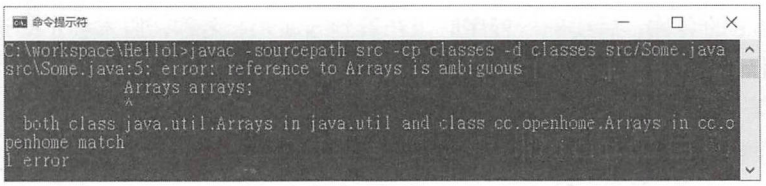


图 2.27　到底是用哪个 Arrays?

当编译程序剖析 Some.java 看到 import 的声明时，会先记得有 cc.openhome 包名称，在继续剖析至 Arrays 该行时，发现它不认识 Arrays 是什么东西，但编译程序记得你用 import 告诉过它。若遇到不认识的名称，可以对比 import 过的名称，编译程序试着将 cc.openhome 与 Arrays 结合在一起为 cc.openhome.Arrays，结果可以在类路径中 cc/openhome 文件夹下找到 Arrays.class。

然而，编译程序试着将 java.util 与 Arrays 结合在一起为 java.util.Arrays，发现也可以在 Java SE API 的 rt.jar 中(默认的类加载路径之一，参考图 2.21)，对应的 java/util 文件夹中找到 Arrays.class，于是编译程序困惑了，到底该使用 cc.openhome.Arrays 还是 java.util.Arrays?

遇到这种情况时，就不能偷懒了，要使用哪个类名称，就得明确地逐字打出来：

```
import cc.openhome.*;
import java.util.*;
public class Some {
    public static void main(String[] args) {
        cc.openhome.Arrays arrays;
    ...
    }
}
```

这个程序就可以通过编译了。简单地说，**import** 是偷懒工具，不能偷懒就回归最保守的写法。

提示»» 学过 C/C++的读者请注意，import 跟#include 一点都不像，无论原始码中有无 import，编译过后的.class 都是一样的，不会影响执行效果。import 顶多只会让编译的时间拉长一些而已。

在 Java SE API 中有许多常用类，像是写第一个 Java 程序时使用的 System 类，其实也有使用包管理，完整名称其实是 java.lang.System。在 **java.lang** 包下的类由于很常用，不用撰写 import 也可以直接使用 class 定义的名称，这也就是为何不用如下撰写程序的原因(写了当然也没关系，只是自找麻烦)：

```
java.lang.System.out.println("Hello World!");
```

如果类位于同一包，彼此使用并不需要 import，当编译程序看到一个没有包管理的类名称，会先在同一包中寻找类，如果找到就使用，如果没找到，再试着从 import 描述进行比对。java.lang 可视为默认就有 import，没有写任何 import 描述时，也会试着比对 java.lang 的组合，看看是否能找到对应类。

2.3　初识模块平台系统

Java SE 9 的重大特性之一是模块平台系统。虽然基于兼容性，你仍然可以使用基于类路径的方式来组织、建立链接库，然而，Java SE 9 的标准 API 已经划分为各个模块，日后在查询 API

文件、使用其他运用了模块的链接库时，总是会面对模块相关信息，因而还是趁早接触比较好。

当然，模块平台系统真要深入，也是相当复杂的，在一开始通盘了解的意义不大，依据遇到的需求逐步认识，对于模块平台系统的认识才会扎实，因而这里会讲到一些基础，之后各章节若需要，再逐步引入更多的信息。

2.3.1 JVM(java)与 module-path

对于模块平台系统要知道的第一件事是，它跟 Java 程序语言本身没有关系，而是为了管理链接库本身的功能封装、链接库之间的相依性等需求而存在。

那么为什么模块可以改进链接库的封装性与相依性？以前面范例的 Console 类为例，你基于它撰写了新的链接库，接着有同事想要使用你的链接库，而你不想要他直接调用 Console 类相关功能，以免他撰写的程序直接依赖在 Console 类。就目前你知道的知识来说，只要类路径上可以找得到 Console 类，他就可以调用相关功能，以后链接库之间错综复杂的相依性就从此开始了，而这也是目前 Java 生态圈中面临的重大问题之一。

当然，Java 生态圈 20 多年来为了这样的问题提出了解决方案，也有第三方(Third-party)的模块系统。为了要统一模块平台的规格，以及为 Java SE 平台“减肥”(让小型运算设备可以依需求下载必要的模块而不是整个 JRE)、考虑改进安全等因素，Java SE 9 决定纳入模块平台系统成为标准之一。

模块平台系统跟 Java 程序语言本身没有关系，而且指令上的运用较为复杂，基本上会有相关开发工具代劳指令处理这些细节。不过，通过一些手动建立模块的过程，有助于了解模块的基本架构，也比较清楚开发工具大致上代劳了哪些细节。

那么就开始建立第一个模块吧！首先看看，一个未支持模块的程序项目要如何设定，才能使之成为模块。请复制范例文件中 labs/CH02 文件夹的 Hello2 文件夹至 C:\workspace，Hello2 中有个 src 文件夹，其中有 cc\openhome\Main.java 档案内容如下：

```
package cc.openhome;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

在进入 Hello2 文件夹后，可以使用下面的指令来编译并执行程序：

```
javac -d classes src/cc/openhome/Main.java
java -cp classes cc.openhome.Main
```

这是基于类路径的方式，若想要设定它为模块，第一件事是决定模块名称，这里假设模块名称为 cc.openhome，建议建立一个与模块名称相同的文件夹，然后根据套件设定的阶层放入原始码，在这里可以在 src 中建立 cc.openhome 文件夹，再将原先 src 中的 cc 文件夹放到 cc.openhome 文件夹之中，如图 2.28 所示。

图 2.28　设定 cc.openhome 模块文件夹

接下来，在 cc.openhome 文件夹中建立一个 module-info.java 并撰写如下内容，如图 2.29 所示。

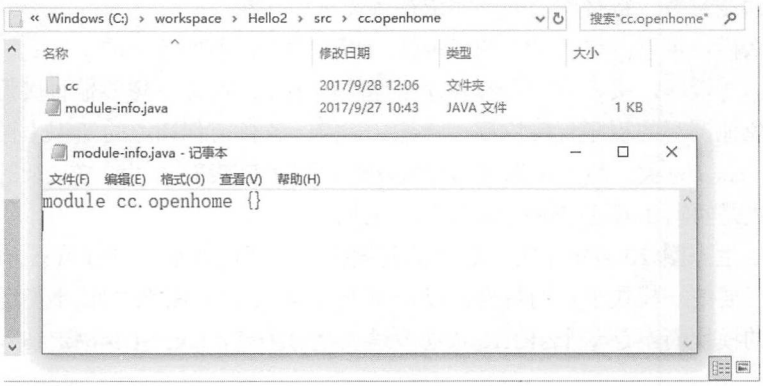


图 2.29　建立 module-info.java

这样，在原始码层面上，建立了第一个模块。虽然 module-info.java 的扩展名为.java，它实际上只是个配置文件，其中的 **module** 关键字仅在这个配置文件中进行设定之用，不是 Java 程序语言的一部分。扩展名为.java，单纯只是为了兼容性，让 javac 等工具程序易于处理这个配置文件罢了。

图 2.29 的 module 关键字定义了模块名称为 cc.openhome，除此之外没有其他设定，这表示目前只能存取同一模块中的 API，并且依赖在 Java 标准 API 的 java.base 模块，java.base 模块包含且 exports 了 java.lang 等常用包(exports 稍后就会介绍)；言下之意也表示，必要的时候，在 module-info.java 中设定自己的模块以公开某些 API，或者是依赖在某个模块之上。

那么来编译程序代码吧！将编译出来的类别放在 mods 文件夹中对应模块名称的文件夹之中，如图 2.30 所示。

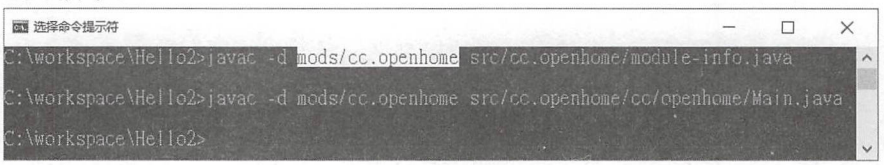


图 2.30　编译模块

这样，mods 中的 cc.openhome 就是你第一个编译成功的模块了，其他开发者若要使用这个模块，可以在执行 **java** 时，通过**--module-path**(或缩写**-p**)指定模块路径。

可以看到，module-info.java 也必须编译，这会产生 module-info.class，在模块封装之后，实际上工具会载入、读取 module-info.class 中的信息，后面的章节谈到模块设定时，会以模块描述文件(Module descriptor)来代称 module-info.java 或 module-info.class。

由于 Main.java 中实际上撰写了程序进入点，如果想要执行它的话，可以通过**--module** 或缩写**-m** 指定模块的程序进入点，如图 2.31 所示。

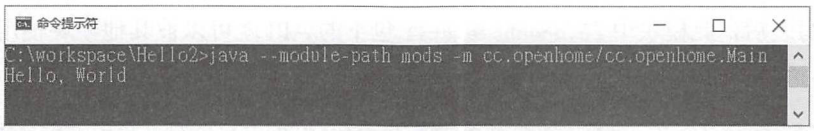


图 2.31　执行模块的程序进入点

在图 2.31 中使用了--module-path 指定了模块路径，而不是使用-classpath 指定类路径，这个时候要注意的是，在完全吻合名称之前还要指定模块名称。然而，这仅仅只是工具层面的需求，在撰写程序代码时，使用模块中的 API 并不进行任何变更。

提示 »» 如果使用 jar 工具程序，使用--main-class 指定程序进入点主类，将模块封装为 JAR 文档，那么-m 就只需要指定模块名称，详情可参考 9.3.1。

cc.openhome.Main 也可以基于类路径来使用，方式与前面小节的介绍是相同的，如图 2.32 所示。

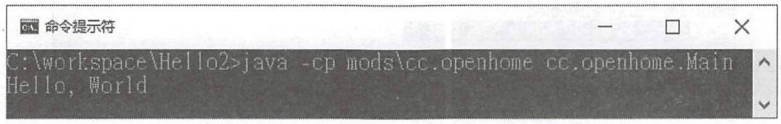


图 2.32　基于兼容性可以指定类路径

按照规范，在类路径下被发现的类，都会被自动归类到一个未命名模块(Unnamed module)。就目前来说你只要知道，基于兼容性，未命名模块可以读取其他模块；相对地，在模块路径下被发现的类别，都属于某个命名模块(Named module)，例如前面--module-path 指定的 mods 中，cc.openhome 模块就是一种命名模块，名称为 cc.openhome。

注意 »» 请不要混用类路径与模块路径，明确定义的模块不能 requires 未命名模块，因为未命名模块没有名称(在 16.1.2 节会说明原因)。

2.3.2　编译程序(javac)与 module-path

假设现在基于某个原因，想要将 2.2.3 节的成果拆分为两个模块，其中 cc.openhome.util 模块将会包含 cc.openhome.util.Console 类，而 cc.openhome 模块将会包含 cc.openhome.Main，并依赖在 cc.openhome.util 模块的 cc.openhome.util.Console 类，最后仍能够执行显示出 Hello, World，该怎么做呢？

此时，你可以如刚才 2.3.1 节建立起对应的文件夹与对应的模块描述文件。不过，这次必须在模块描述文件做设定，为了练习时的方便，可以直接复制范例文件中 labs/CH02 的 Hello3 文件夹至 C:\workspace，其中已经建立好两个模块应有的文件夹及模块描述文件，不过模块描述文件都还没有任何额外的设定。

先来处理 cc.openhome.util 模块，为了让其他模块能使用此模块下的 API，必须在模块描述文件中使用 **exports** 声明，声明哪些包中的公共(**public**)类、方法或值是可以存取的，因此打开 Hello3/src/cc.openhome.util/module-info.java 文件，并进行图 2.33 所示设定。

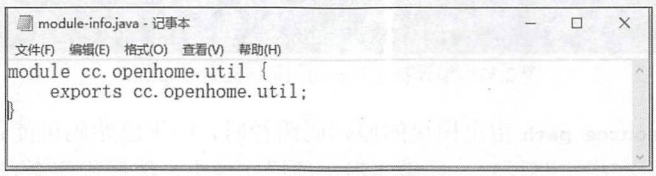


图 2.33　公开模块中的套件

这样撰写之后，此模块中 cc.openhome.util 包下的 API 就可以被其他模块使用了，现在对 cc.openhome.util 模块进行编译，如图 2.34 所示。

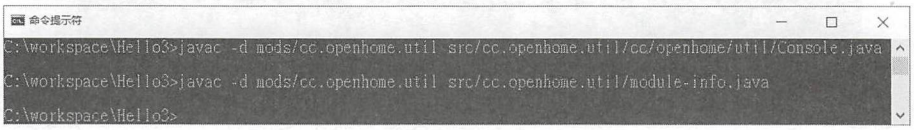


图 2.34　编译 cc.openhome.util 模块

接下来，因为 cc.openhome 模块会依赖在 cc.openhome.util 模块，所以必须在 cc.openhome 模块的模块描述文件使用 **requires** 声明依赖的模块，打开 Hello3/src/cc.openhome/module-info.java，并进行图 2.35 所示设定。

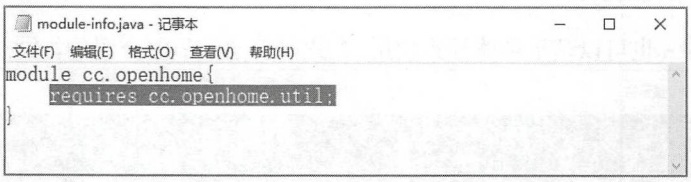


图 2.35　依赖于 cc.openhome.util 模块

接下来可以对 cc.openhome 模块进行编译，然后执行显示 Hello, World，如图 2.36 所示。

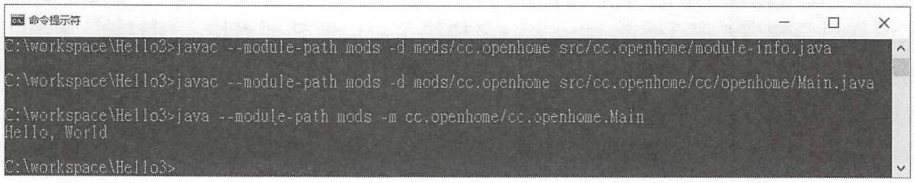


图 2.36　编译与执行 cc.openhome 模块

可以看到，在使用 **javac** 进行编译时，也可以使用**--module-path** 指定模块路径，模块路径下各模块的模块描述文件，包括了模块中 API 的依赖以及是否存取关系，以此决定是否顺利通过编译。

2.3.3　编译程序(javac)与 module-source-path

你也许会想问，在 2.2.1 节曾经谈过使用 javac 编译时，可用-sourcepath 指定原始码路径，那么在编译模块时有类似的自变量吗？因为也许你拿到了其他模块的原始码(而不是编译或进一步封装好的模块)，想要自行编译出.class 文件。

在使用 javac 时，确实有个--module-source-path 可以指定模块的原始码路径，以刚才的范例来说，若不想分别对 cc.openhome.util 与 cc.openhome 模块分别进行编译，那么只要指定 **--module-source-path** 就可以了，如图 2.37 所示。

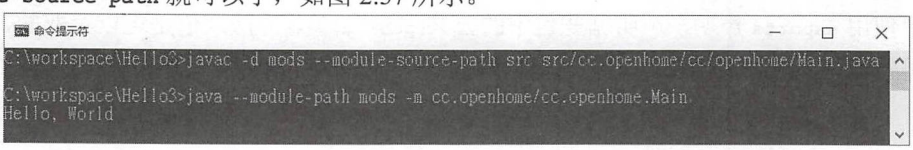


图 2.37　编译指定--module-source-path

使用**--module-source-path** 指定模块的原始码路径时，由于原始码可能是来自于多个模块，因此搭配的-d 自变量在指定路径时，只需要指定顶层文件夹，编译程序会自行建立起对应于模

块名称的文件夹。

实际上，在能够运用--module-path 与--module-source-path 自变量之后，搭配-d 自变量时，本来就只需指定顶层文件夹，2.3.1、2.3.2 节只是循序渐进的过程，让你逐步了解--module-path 与--module-source-path 的作用，也就是说，编译 cc.openhome.util 及 cc.openhome 模块就可以了，如图 2.38 所示。

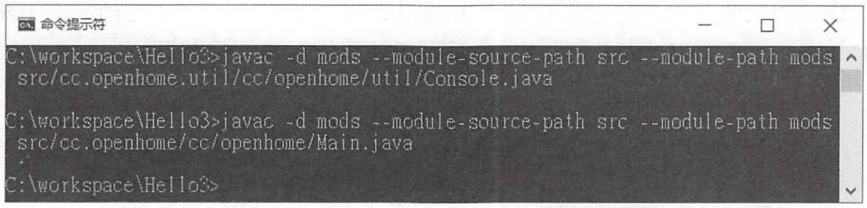


图 2.38 编译指定--module-source-path 与--module-path

简单来说，可以暂时将--module-path 与--module-source-path 想象成-classpath 与-sourcepath 的对应物，这样在指定自变量时就容易记忆且不会产生混淆。

就初学而言，对模块的认识暂时到这里就足够了，在之后的章节若有必要，会适当引入其他模块在使用上的信息，而在第 19 章，也会更详细地进行模块的介绍。

提示 »» 如果这些对模块的基本认识还不过瘾，也可以先看看 *Project Jigsaw: Module System Quick-Start Guide*：
http://openjdk.java.net/projects/jigsaw/quick-start

2.4 使用 IDE

在开始使用包管理原始码文档与位码文档之后，必须建立与包对应的实体文件夹层级，编译时必须正确指定-sourcepath、-cp 等自变量，执行程序时必须使用完全吻合名称。而在 Java SE 9 支持模块化之后，还必须会使用--module-source-path、--module-path 等自变量，这实在是很麻烦。可以考虑开始使用 IDE(Integrated Development Environment，集成开发环境)，由 IDE 代劳一些原始码文档与位码文档等资源管理工作，提升你的效率。

提示 »» 除了 IDE 之外，也可以考虑使用 Ant 或 Maven 等工具提高效率，参考一下 JUnit 文件中有关 Ant 或 Maven 的介绍：
http://openhome.cc/Gossip/JUnit/
Gradle 的话可以参考一下文件《认识 Gradle》中的介绍：
http://www.codedata.com.tw/java/understanding-gradle-1-ant/

2.4.1 IDE 项目管理基础

在 Java 领域中，有为数不少的 IDE 是优秀的开放原始码产品，最为人熟知的 IDE 有 NetBeans、Eclipse、Intellij IDEA、JDeveloper 等。不同的 IDE 会有不同的特色，但基本概念通常相同。最重要的是，只要了解 JDK 与相关指令操作，就不容易被特定的 IDE 给限制住。

在本书中，将选择 NetBeans IDE 进行基本介绍。选择 NetBeans 的原因在于，NetBeans 直接使用用户安装的 JDK，而 IDE 上显示的编译错误信息就是 JDK 实际显示的信息，这对初学

者了解 JDK 与 IDE 功能对应有帮助。

NetBeans 的官方网址为 netbeans.org，在撰写本书的时候，NetBeans 最新正式版本是 8.2，然而不支持 JDK9。不过，最新的 Development Build 已经支持 JDK9，就本书范围而言，只要下载 Java SE 版本即可：

bits.netbeans.org/download/trunk/nightly/latest/

执行下载后的安装文件并同意授权后，出现图 2.39 所示画面时，请选择你的 JDK9 安装目录，例如 C:\Program Files\Java\jdk-9.0.4，之后逐步按照指示进行安装即可。

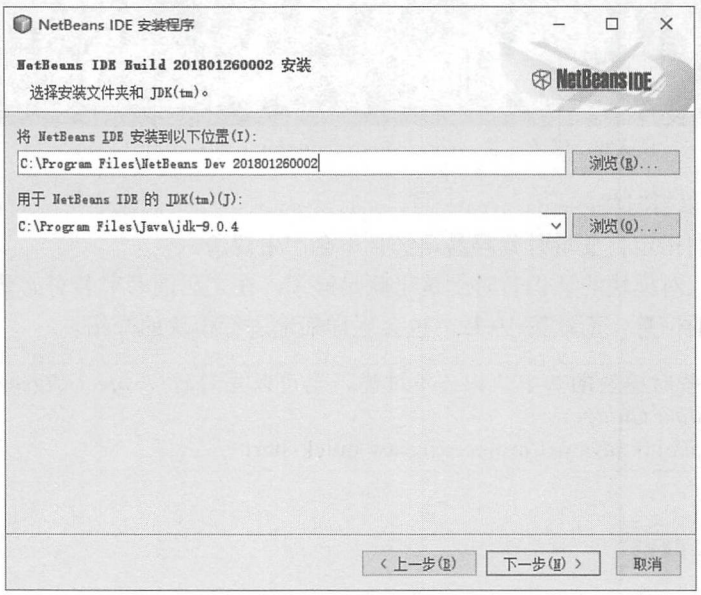


图 2.39　NetBeans IDE 直接使用你安装的 JDK

提示 »» 如果 Netbeans 无法安装，提示找不到 JDK。可以用 JAVA_HOME 来指定 JDK 位置。步骤如下：

(1) 打开 CMD，进入 Netbeans 安装文件所在目录。

(2) 在 CMD 里启动 Netbeans 安装文件，后面加上参数 --javahome JDK 目录。例如：F:\>netbeans-trunk-nightly-201801260002-windows.exe --javahome "C:\Program Files\Java\jre-9.0.4"

在程序规模步入必须使用包管理之后，就等于初步开始了项目资源管理。在 IDE 中要撰写程序，通常也是从建立项目开始。在 IDE 中会提供多种项目样板，这里先来介绍基本的 Java Application，启动 NetBeans 后，可以按如下步骤建立 Java Application。

(1) 选择“文件”|“新建项目”命令，在弹出的“新建项目”对话框的“类别”列表中选择 Java，在“项目”列表中选择“Java 应用程序”，接着单击“下一步”按钮，如图 2.40 所示。

(2) 在“项目名称”文本框中输入项目名称 Hello4，在“项目位置”文本框中输入 C:\workspace，注意，“项目文件夹”会保存至 C:\workspace\Hello4，如图 2.41 所示。

(3) 在“创建主类”文本框中输入 cc.openhome.Main，这表示会有个 `Main` 类放在 `cc.openhome` 包，其中会自动建立 `main()` 程序进入点方法，接着单击“完成”按钮建立项目。

图 2.40　新建项目

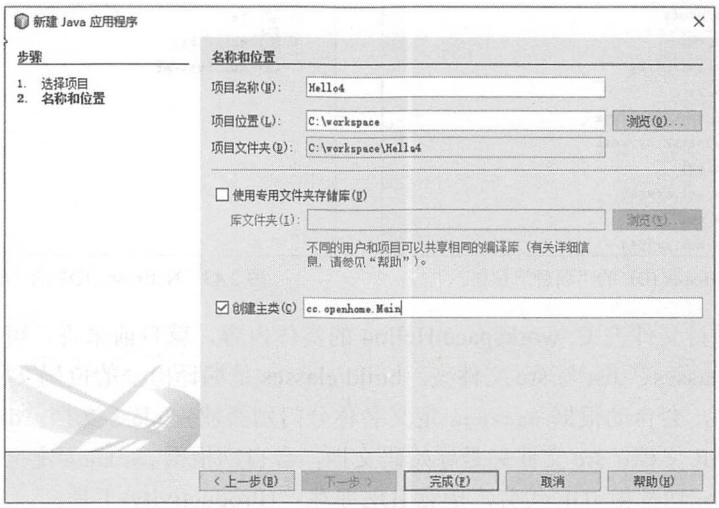


图 2.41　设置项目名称和位置

项目建立后，IDE 通常会提供默认的项目检查窗格，方便用户检视一些项目资源。NetBeans 会提供图 2.42 所示的“项目”窗格。

在图 2.42 所示的“项目”窗格中，可以看到“源包”中包含 cc.openhome，其中放置了 Main.java，这方便用户以包为单位查看原始码文档。而在“库”中可看到使用了 JDK9 中的一些模块，Java Application 默认并不支持模块化，会采用类路径方式，不理会各模块中模块描述文件设定，因而可以使用整个 Java SE 标准 API。

下面会看到，如果你的项目是基于 JDK8 或更早版本的话，由于旧版本并不支持模块化，因而在“库”中看到的是一堆 JAR 文件，表示 IDE 管理的 CLASSPATH 会包括这些 JAR 文件。在“库”上右击，将相应的程序 JAR 文件(或其他项目)加入至目前的项目之中，IDE 会自动管理 CLASSPATH 细节。

在 Main.java 的 Main() 方法中进行如下撰写，然后执行“运行”|“构建项目”命令，要求 NetBeans 进行程序编译：

```
System.out.println("Hello World");
```

“项目”窗格提供方便的项目资源查看，但不等于实体文档管理，有的 IDE 必须自行打开“资源管理器”来查看项目文件夹内容，NetBeans 则可以切换至“文件”窗格直接查看实际文档管理，如图 2.43 所示。

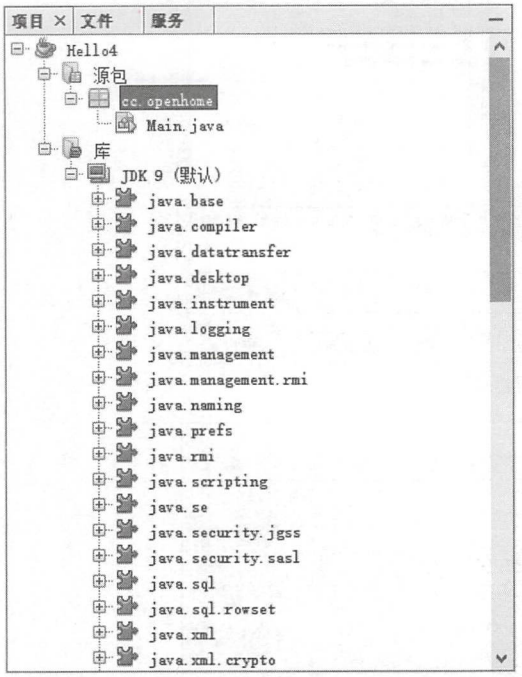


图 2.42 NetBeans IDE 的“项目”窗格

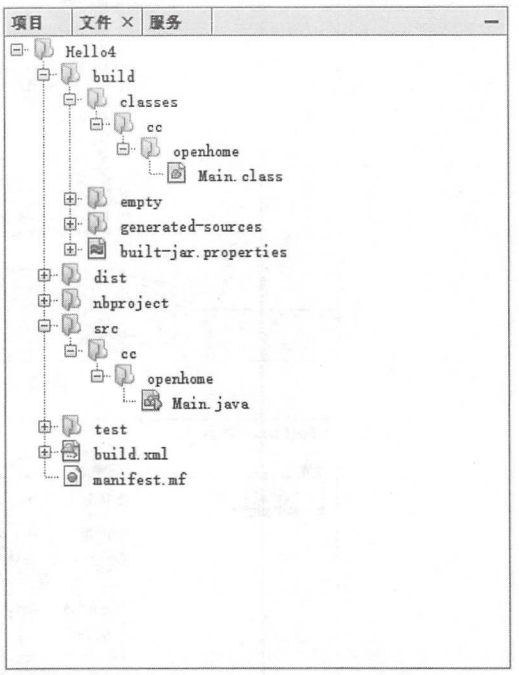


图 2.43 NetBeans IDE 的“文件”窗格

图 2.43 是项目文件夹 C:\workspace\Hello4 的实体内容，就目前来说，可先注意“文件”窗格中的 build/classes、dist 与 src 文件夹。build/classes 是编译出来的位码文档(在执行时会用到的 `CLASSPATH`)，会自动根据 `package` 定义名称分门别类放置.class 文档；dist 文件夹是封装了位码文档的 JAR 文档；src 文件夹是原始码文档，会自动根据 `package` 定义名称分门别类放置.java 文档。这一切都由 IDE 代劳，毕竟 IDE 是生产(Productivity)工具。

如果要使用 NetBeans 执行程序进入 `main()` 的类，可右击 Main.java 文件，在弹出的快捷菜单中选择“运行文件”命令，会有个“输出”窗口显示执行结果，如图 2.44 所示。

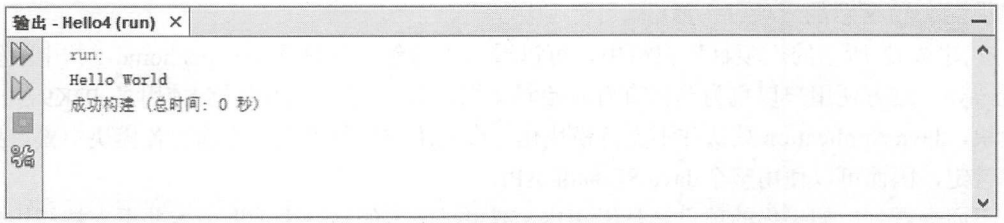


图 2.44 NetBeans IDE 的“输出”窗口

在 IDE 中编辑程序代码，若出现红色虚线，通常表示那是导致编译错误的语法。如果看到红色虚线，把光标移至红色虚线上，就会显示编译错误信息，如果是 NetBeans，会直接显示 JDK 编译工具提供的错误信息。例如，图 2.45 所示是 Main.java 中 `public class` 定义的名称不等于 Main(主文档名)而产生的编译错误与信息。

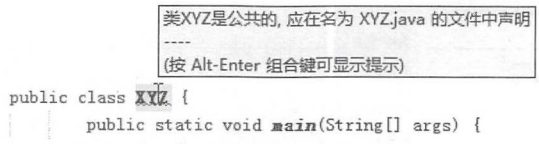


图 2.45　在 IDE 中，红色虚线通常表示编译错误

对于一些编译错误，IDE 也会提示一些改正方式。以 NetBeans 来说，会出现一个小电灯炮图标，这时可以单击图标显示改正提示，看看是否有合用的选项，如图 2.46 所示。

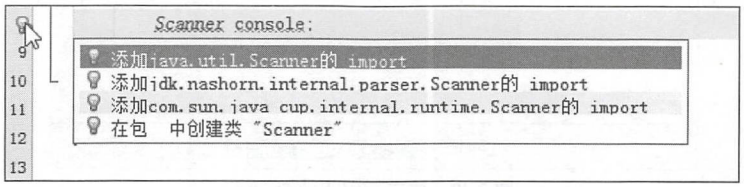


图 2.46　编译错误时的改正提示

以图 2.46 为例，编译程序不认识 `Scanner` 类而发生编译错误，第一个提示选项是 IDE 发现有个 `java.util.Scanner` 也许是你想要的，看你是不是要 `import`。你也可以看到，在其他包分类中也有 `Scanner`，另外还有建立类的选项。IDE 有提示是好事，但还是要判断哪个选项才是自己想要的，不是单击第一个选项就没事了。

使用 NetBeans 的“Java 应用程序”建立项目的话，即使使用了 JDK9，默认也不支持模块化，有个简单的方式可以让它支持模块管理。操作步骤如下：

(1) 在 Hello4 图标上右击，在弹出的快捷菜单中选择“新建”|“其他”命令，将出现“新建文件”对话框，在“类别”列表中选择 Java，在“文件类型”列表中选择 Java Module Info，然后单击“下一步”按钮，如图 2.47 所示。

图 2.47　创建 Java Module Info

(2) 在 New Java Module Info 对话框中，可以在“创建的文件”文本框中看到将建立的 module-info.java 文件，单击“完成”按钮进行创建，如图 2.48 所示。

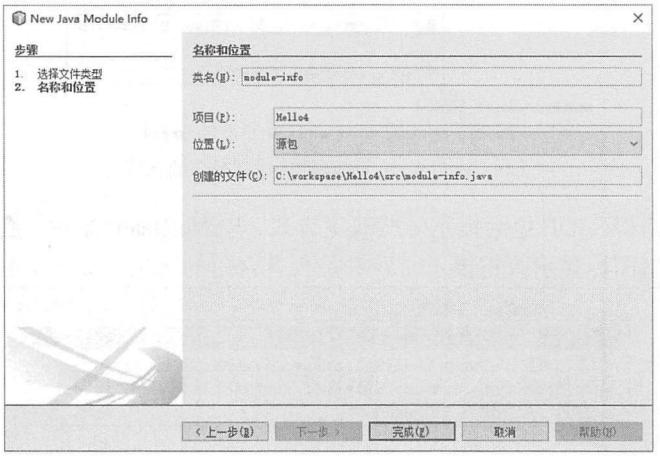


图 2.48　建立 module-info.java

建立的 module-info.java 中，会使用项目名称作为模块名称，也就是会有 `module Hello4 {}` 的内容，这表示仅依赖在 `java.base` 模块，在“项目”窗格的“库”中可以看到只剩下一个 `java.base` 模块，如图 2.49 所示。

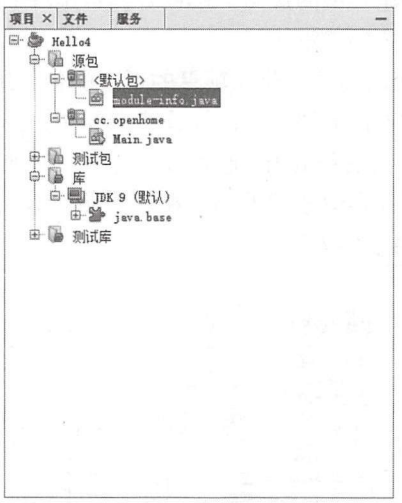


图 2.49　默认依赖在 `java.base` 模块

试着创建其他所需的 module-info.java 模块，在“库”中会有对应的模块图标出现，今后若在“库”中加入新的依赖模块，module-info.java 中也会自动撰写必要的设定。

如果在新建项目时，一开始就打算支持模块化，NetBeans 提供了 Java Module Project，你可以按照前面介绍的新建项目的步骤，在“新建项目”对话框的“类别”列表中选择 Java，在“项目”列表中选择 Java Module Project，然后单击“下一步”按钮，将项目命名为 Hello5，然后单击“完成”按钮创建项目。

在建立项目后，在 Hello5 图标上右击，在弹出的快捷菜单中选择“新建”| Module 命令，就可以建立新模块了，如图 2.50 所示。

在接下来的 Name Module 对话框中，可以为模块命名(如 `cc.openhom`)，在建立模块之后，右击模块图标，在弹出的快捷菜单中选择“新建”|“Java 类”命令来建立类，如建立一个 `cc.openhome.Main` 类，“项目”窗格如图 2.51 所示。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

图 2.50　在 Java Module Project 建立新模块

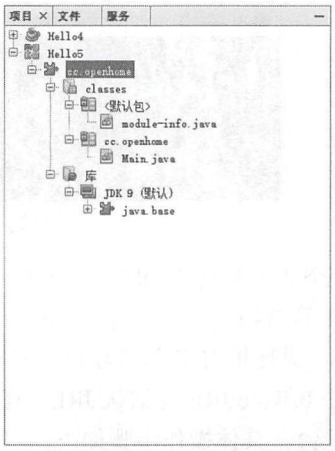


图 2.51　建立的 cc.openhome 模块

按照相同的方式，在这个 Hello5 项目中建立多个模块，本书后续也将会采用 Java Module Project 来建立范例项目。

在采用了 IDE 进行开发之后，一些指令、项目文件夹结构或 JAR 文件的建立等细节，都由 IDE 代劳了，因而可以将注意力集中在包、类的程序代码撰写以及模块描述文件的设定等重要的关注点上。

以上简单解释了 CLASSPATH、JDK 工具使用、编译相关错误信息、包管理、模块等概念，对应到 IDE 中哪些操作或设定以及其他的功能，会在之后说明相关内容时一并介绍。

2.4.2　使用了哪个 JRE

因为各种原因，你的计算机中可能不只存在一套 JRE。可以试着搜索计算机中的文档，例如在 Windows 中搜索 java.exe，可能会发现有多个 java.exe 文档。在某些程度上而言，可以将找到一个 java.exe 视作就是有一套 JRE。

在安装好 JDK 后，如果选择一并安装 Public JRE，至少会有两套 JRE 存在计算机中，一个是 JDK 本身的 Private JRE，一个是选择安装的 Public JRE。

既然计算机中有可能同时存在多套 JRE，那么你到底执行了哪一套 JRE？在文本模式下输入 java 指令，如果设定了 PATH，会执行 PATH 顺序下找到的第一个 java 可执行文件，这个可执行文件所启动的是哪套 JRE？

如果设定 PATH 包括 JDK 的 bin 目录，执行 java 指令时，会使用 JDK 的 bin 中的 java(.exe)，因而找到的会是 JDK 的 Private JRE。

如果将 PATH 设定包括 Public JRE 的文件夹，例如 C:\Program Files\Java\jdk-9.0.4\bin，则在执行 java 指令时，使用的就是 JRE 的 bin 中的 java(.exe)，所以使用 C:\Program Files\Java\jdk-9.0.4\bin 这个 Public JRE。

在执行 java 指令时，可以附带一个-version 变量，这可以显示执行的 JRE 版本，这是确认所执行 JRE 版本的一个方式，如图 2.52 所示。

在刚到一个新开发环境时(例如到客户那边去时)，先确认版本是很重要的一件事。文本模式下若要确认 JRE，可先检查 PATH 路径中的顺序，再查看 java-version 的信息，这些都是基本的检查动作。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

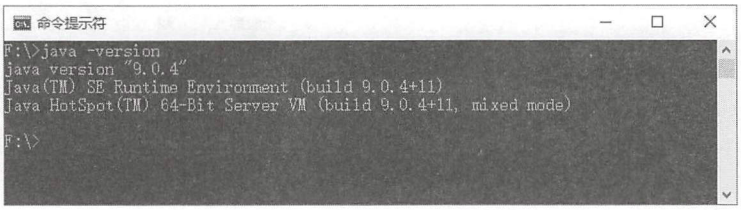


图 2.52　使用-version 确认版本

如果有个需求是切换 JRE，文本模式下必须设定 PATH 顺序中找到的第一个 JDK 或 JRE 的 bin 文件夹，这就决定了要使用的 JRE。

那么，如果使用 IDE 新建项目，你使用了哪个 JRE 呢？如果是 NetBeans，会以你安装时设定的 JDK 中 Private JRE 为默认 JRE。在 NetBeans 中，如果想切换所使用的 JDK(JRE)，可以先新建 Java 平台。具体操作步骤如下：

(1) 选择“工具”|“Java 平台”命令，打开“Java 平台管理器”对话框，单击“添加平台”按钮。

(2) 在打开的“添加 Java 平台”对话框中，选择 Java Standard Edition 单选按钮，单击“下一步”按钮。

(3) 选择你想要的 JDK 文件夹(例如 JDK9 的文件夹)，单击“下一步”按钮。

(4) 确认默认的“平台名称”“平台源”和“平台 Javadoc”是你想要的设定值后，单击“完成”按钮完成平台添加，如图 2.53 所示。

(5) 在“Java 平台管理器”对话框中单击“关闭”按钮完成设定。

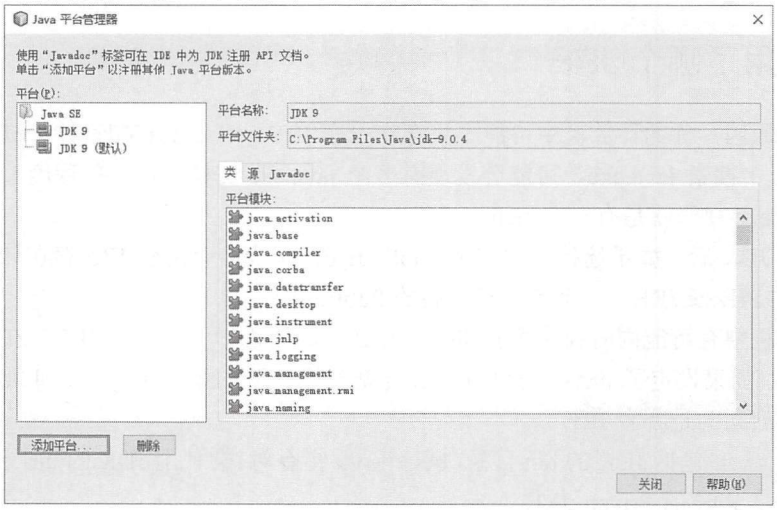


图 2.53　管理 Java 平台

完成 Java 平台建立后，接下来可根据以下操作，新增一个 Java 应用程序项目，默认会采用图 2.53 被标示为默认的平台，可以进行如下操作，改变项目想使用的 JDK(JRE)。

(1) 在“项目”窗格中选中项目并右击，在弹出的快捷菜单中选择“属性”命令。

(2) 打开“项目属性”对话框，选择“库”选项，在右边的“Java 平台”下拉列表框中选择要使用的 JDK 版本，单击“确定”按钮完成设定。

(3) 在“项目”窗格中的“库”选项下，已设定为你想要的 JDK 版本。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

2.4.3 类文档版本

如果使用新版本 JDK 编译出位码文档，在旧版本 JRE 上执行，有可能会发生图 2.54 所示的错误信息。

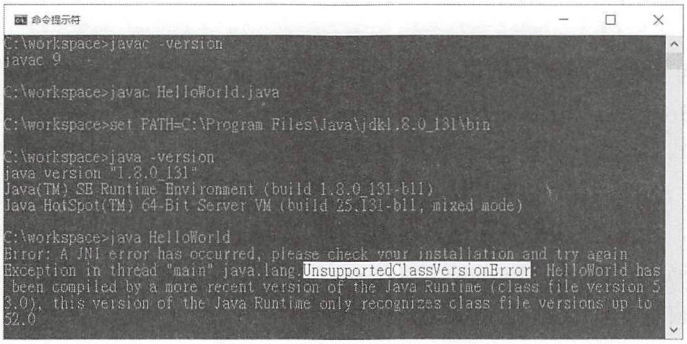


图 2.54　不支持此版本

图 2.54 所示是在 JDK9 下编译，切换 PATH 至 JDK8，使用 Private JRE8 执行位码，结果出现 UnsupportedClassVersionError，并指出这个位码的主版本号与次版本号(major.minor)为 53.0。

编译程序会在位码文档中标示主版本号与次版本号，不同版本号的位码文件格式可能有所不同。JVM 在加载位码文档后，会确认其版本号是否在可接受的范围，否则就不会处理该位码文档。

可以使用 JDK 工具程序 javap 加上-v 或-verbose，确认位码文档的版本号，如图 2.55 所示。

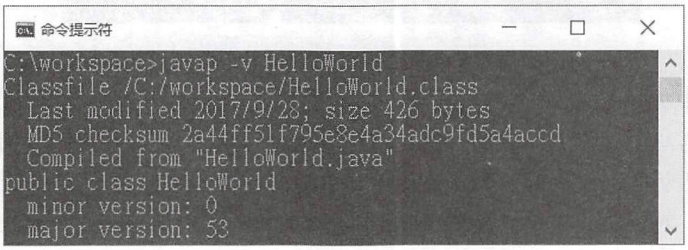


图 2.55　使用 javap 剖析位码文档

可以使用 **System.getProperty("java.class.version")** 取得 JRE 支持的位码版本号，使用 **System.getProperty("java.runtime.version")** 取得 JRE 版本信息。

在编译的时候，可以使用-target 指定编译出来的位码，必须符合指定平台允许的版本号，使用-source 要求编译程序检查使用的语法不超过指定的版本，如图 2.56 所示。

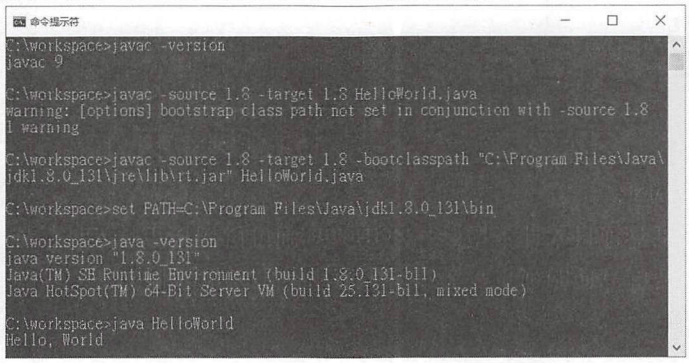


图 2.56　指定-source 与-target 选项

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

上面这个例子指定编译出来的位码文档必须是 1.8 平台可接受的版本号，并检查使用语法必须是 1.8 语法。在不指定-target 与-source 的情况下，编译程序会有默认的-target 值。例如，JDK9 默认的-target 与-source 都是 1.9，-target 在指定时，值必须大于或等于-source，所以在图 2.56 中，若只指定-target 为 1.8，就会无法通过编译，因为-source 仍是默认值 1.9。

从图 2.56 中可看到，如果只指定-source 与-target 进行编译，会出现警示信息，这是因为编译时默认的 Bootstrap 类加载器(Class loader)没有改变，第 17 章会介绍 Bootstrap 类加载器是什么。简单来说，系统默认的类加载器仍会从 Java SE 9 的模块中加载相关类，如果引用到一些旧版 JRE 没有的新 API，就会造成在旧版 JRE 上无法执行，最好是编译时指定-bootclasspath，参考至旧版的 rt.jar，这样在旧版 JRE 执行时不容易发生问题。

由于要同时指定-source、-target 与-bootclasspath 很麻烦，在 JDK9 中使用 javac 时，可以使用**--release** 自变量一次指定，如图 2.57 所示。

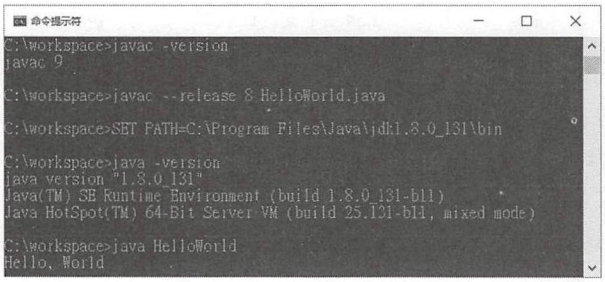


图 2.57　指定--release 选项

那么在 IDE 中如何设定-source 与-target 对应的选项呢？不同版本 IDE 中建立的项目，默认的-source 与-target 选项并不相同。以 NetBeans 为例，如果默认使用 JDK9 的-source 与-target，要想改变为 JDK8，可以在项目上右击，在弹出的快捷菜单中选择“属性”命令，打开“项目属性”对话框，在“类别”列表中选择“源”，在“源/二进制格式”列表框中选择 JDK8，如图 2.58 所示。

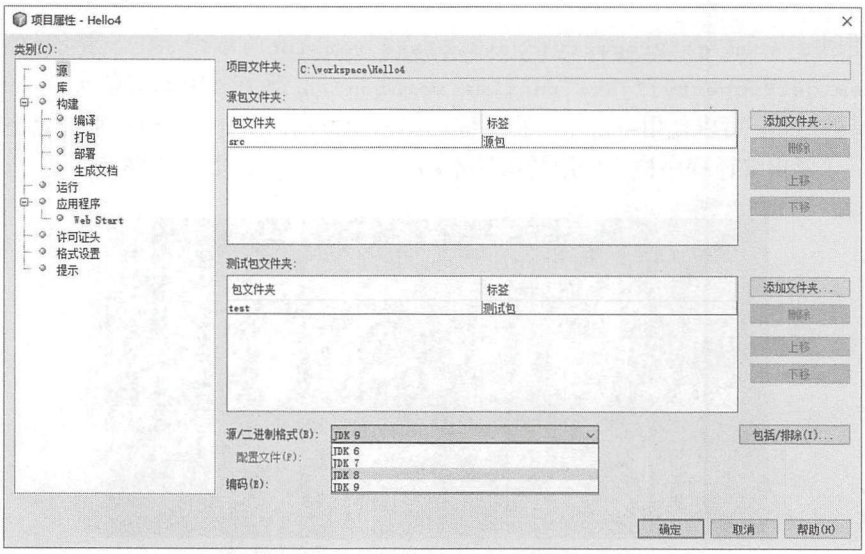


图 2.58　NetBeans 中设定 JDK 的-source 与-target

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

2.5 重点复习

在正式撰写程序之前，请先确定你可以看到文档的扩展名。撰写 Java 程序时有几点必须注意：

- 扩展名是.java。
- 主文档名与类名称必须相同。
- 注意每个字母大小写。
- 空格只能是半角空格符或 Tab 字符。

一个.java 文档可定义多个类，但是只能有一个公开(public)类，而且主文档名必须与公开类名称相同。规格书中规定 main() 方法的形式一定得是：

```
public static void main(String[] args)
```

当输入一个指令而没有指定路径信息时，操作系统会依照 PATH 环境变量中设定的路径顺序，依序寻找各路径下是否有这个指令。若系统中安装两个以上 JDK，Path 路径中设定的顺序将决定执行哪个 JDK 下的工具程序。在安装了多个 JDK 或 JRE 的计算机中，确定执行了哪个版本的 JDK 或 JRE 非常重要，确定 PATH 信息是一定要做的动作。

在 JVM 中执行某个可执行文件(.class)，就要告诉 JVM 这个虚拟操作系统到哪些路径下寻找文档，在 JDK8 或更早版本中，方式是通过 CLASSPATH 指定可执行文件(.class)的路径信息。在启动 JVM 时要告知可执行文件(.class)的位置，可以使用-classpath 或-cp 自变量来指定。如果希望从目前文件夹开始寻找类文档，则可以使用“.”指定。

JAR 文档实际使用 ZIP 格式压缩，当中包含一堆.class 文档，设定 CLASSPATH 时可将 JAR 文档当作特别的文件夹。如果某个文件夹中有许多.jar 文档，从 Java SE 6 开始，可以使用“*”表示使用文件夹中所有.jar 文档。

在使用 javac 编译程序时，如果要使用到其他类链接库，在 JDK8 或更早版本中，必须使用-cp 指定 CLASSPATH，使用-sourcepath 指定寻找原始码文档的文件夹，使用-d 指定编译完成的位码存放文件夹，指定-verbose 自变量可看到编译程序进行编译时的过程。

当类原始码开始使用 package 进行分类时，就会具有以下管理上的意义：

- 原始码文档要放置在与 package 所定义名称层级相同的文件夹层级。
- package 所定义名称与 class 所定义名称，会结合而成类的完全吻合名称(Fully Qualified Name)。
- 源文件要放置在与 package 所定义名称层级相同的文件夹层级。
- 在包间可以直接使用的类或方法(Method)必须声明为 public。

import 只是偷懒工具，让你在原始码中不用使用完全吻合名称。

对于模块平台系统要知道的第一件事是，它跟 Java 程序语言本身没有关系。虽然 module-info.java 的扩展名为.java，它实际上只是个配置文件，其中的 module 关键字仅在这个配置文件中进行设定之用，不是 Java 程序语言的一部分。

如果模块描述文件中没有任何需要设定，目前只能存取同一模块以及 Java 标准 API 的 java.base 模块。

在完全吻合名称之前还要指定模块名称，然而，这仅仅只是工具层面的需求，在程序代码

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

撰写上，使用模块中的 API 并不用进行任何变更。

基于兼容性，模块也可以基于类别路径来使用。在类别路径下被发现的类，都会被自动归类到一个未命名模块。就目前来说，只要知道未命名模块可以读取其他模块；相对地，在模块路径下被发现的类，都属于某个命名模块。

为了让其他模块能使用此模块下的 API，必须在模块描述文件中使用 `exports` 声明，声明哪些包中的公共(`public`)类、方法或值是可以存取的。模块描述文件中必须使用 `requires` 声明依赖的模块。

可以暂时将`--module-path` 与`--module-source-path` 想象成`-classpath` 与`-sourcepath` 的对应物，这样在指定自变量就容易记忆且不会产生混淆了。

如果设定 `PATH` 包括 JDK 的 bin 文件夹，执行 `java` 指令时，会使用 JDK 的 bin 中的 java(.exe)，因而找到的会是 JDK 的 Private JRE。

如果将 `PATH` 设定包括 Public JRE 的文件夹，如 C:\Program Files\Java\jdk-9.0.4\bin，则执行 `java` 指令时，使用的就是 JRE 的 bin 中的 java(.exe)，所以使用 C:\Program Files\Java\jre-9.0.4\ 这个 Public JRE。

在执行 `java` 指令时，可以附带一个`-version` 自变量显示执行的 JRE 版本。在编译的时候，可以使用`-target` 指定编译出来的位码，必须符合指定平台所允许的版本号，使用`-source` 要求编译程序检查使用的语法，不超过指定的版本。JDK9 默认的`-target` 与`-source` 都是 1.9，`-target` 在指定时，值必须大于或等于`-source`。

由于要同时指定`-source`、`-target` 与`-bootclasspath` 很麻烦，在 JDK9 中使用 `javac` 时，可以使用`--release` 自变量一次指定。

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 3 基础语法

学习目标

- 认识类型与变量
- 学习运算符的基本使用
- 了解类型转换细节
- 运用基本流程语法
- 使用 jshel

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

3.1 类型、变量与运算符

Java是个支持面向对象的程序语言，但在正式进入面向对象支持语法的介绍前，读者对于类型、变量、运算符、流程控制等这些各种程序语言的基本语法元素，要有一定的了解。虽然各种程序语言都有这些基本语法元素，但别因此而忽视它们，因为各种程序语言都有其诞生的宗旨与演化过程，对这些基本语法元素也就会有其独有的特性。

本书先从类型开始介绍。

3.1.1 类型

在Java的世界中，并非每个东西都可抽象化为对象，所以还是要面对系统的一些特性。例如，关于内存长度有限的问题，在程序执行时，遇到123这个整数，还是要想一下，用多少长度的内存来储存它会比较经济。基本上，Java可区分为基本类型(Primitive Type)和类类型(Class Type)两大类型系统，其中，类类型也称为参考类型(Reference Type)。

本章先介绍基本类型，在第4章介绍类类型。

所谓基本类型，就是在使用时，得考虑一下数据用多少内存长度来存比较经济，利用程序语法告诉JVM，然后由JVM自动为用户在内存中配置与管理。在Java中的基本类型主要可区分为整数、字节、浮点数、字符与布尔。

- 整数：可细分为 **short** 整数(占2字节)、**int** 整数(占4字节)与 **long** 整数(占8字节)。不同长度的整数，可储存的整数范围也不同。long整数占的内存长度比 int 整数来得多，可表示的整数范围也就比 int 整数大。而 int 整数可表示的整数范围也比短整数来得大。
- 字节：**byte** 类型顾名思义，长度就是1字节，在需要逐字节处理数据时(如图像处理、编码处理等)，就会使用 byte 类型，若用于表示整数，byte 可表示-128~127的整数。
- 浮点数：主要用来储存小数数值，可分为 float 浮点数(占4字节)与 double 浮点数(占8字节)。double 浮点数使用的内存空间比 float 浮点数来得多，可表示的精确度也比较大。
- 字符：**char** 类型用来储存'A'、'B'、'林'等字符符号。在JDK9中，Java的字符采用Unicode 8.0编码，JVM结果采用UTF-16 Big Endian，所以每个字符类型占2字节，汉字字符与英文字符在Java中同样都是用双字节储存。

提示>>> 编码？Unicode？UTF？建议先阅读以下文件(《哪来的纯文本档？》和《Unicode与UTF》)：
http://openhome.cc/Gossip/Encoding/TextFile.html
http://openhome.cc/Gossip/Encoding/UnicodeUTF.html

- 布尔：**boolean** 类型可表示 **true** 与 **false**，分别代表逻辑的“真”与“假”。在Java中不用在意 boolean 类型的长度，因为 boolean 类型与其他类型无法做运算。

每种类型占有的内存长度不同，可储存的数值范围也就不同。例如，int 类型的内存空间

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

是 4 字节，所以可储存的整数范围为-2 147 483 648~2 147 483 647，如果储存值超出类型范围，称为溢值(Overflow)，会造成程序不可预期的结果。不用记忆各种类型可储存的数值范围，可以通过 API 来得知。例如：

Basic_Range.java

```
package cc.openhome;
public class Range {
    public static void main(String[] args) {
        // byte、short、int、long 范围
        System.out.printf("%d ~ %d%n", Byte.MIN_VALUE, Byte.MAX_VALUE);
        System.out.printf("%d ~ %d%n", Short.MIN_VALUE, Short.MAX_VALUE);
        System.out.printf("%d ~ %d%n", Integer.MIN_VALUE, Integer.MAX_VALUE);
        System.out.printf("%d ~ %d%n", Long.MIN_VALUE, Long.MAX_VALUE);
        // float、double 精度范围
        System.out.printf("%d ~ %d%n", Float.MIN_EXPONENT, Float.MAX_EXPONENT);
        System.out.printf("%d ~ %d%n", Double.MIN_EXPONENT, Double.MAX_EXPONENT);
        // char 可表示的 Unicode 范围
        System.out.printf("%h ~ %h%n", Character.MIN_VALUE, Character.MAX_VALUE);
        // boolean 的两个值
        System.out.printf("%b ~ %b%n", Boolean.TRUE, Boolean.FALSE);
    }
}
```

对于初学 Java 者，会看到一些新的语法与 API 在里面，以下逐一介绍。

你在程序中看到“//”符号，这是 Java 程序中的单行批注。批注是用来说明或记录程序中一些注意事项，编译程序会忽略该行“//”符号之后的文字，对编译出来的程序不会有任何影响。另一个批注符号是“/*”与“*/”包括的多行批注。例如：

```
/* 作者：良葛格
   功能：示范 printf()方法
   日期：20117/10/1
 */
public class Demo {
...
```

编译程序会忽略“/*”与“*/”间的文字，不过以下使用多行批注的方式是不对的：

```
/*  批注文字 1…bla…bla
    /*
        批注文字 2…bla…bla
    */
*/
```

编译程序会以为倒数第二个“*/”就是批注结束的时候，因而对最后一个“*/”就会认为是错误的语法，这时就会出现编译错误的信息。

`System.out.printf()`是标准 API，在第一个 Java 程序显示 Hello World 时，使用了 `System.out.println()`，这会在标准输出中显示文字后换行。如果使用 `System.out.print()`，则输出文字后不会换行。那么，`System.out.printf()`是什么？

`printf()`是 JDK5 之后才有的 API，`f` 是 format 的意思，也就是格式化，用在 `System.out` 上，就是对输出文字做格式化后再显示在文本模式中。`printf()`的第一个自变量(Argument)是字符串，其中`%d`、`%h`、`%b` 等是格式控制符号。表 3.1 列出了一些常用的格式控制符号。

严禁网络传播本PDF，违者责任自负！

表 3.1 常用格式控制符号

符 号	说 明
%%	因为%符号已经被用来作为控制符号前置，所以规定使用%%才能在字符串中表示%
%d	以十进制整数格式输出，可用于 byte、short、int、long、Byte、Short、Integer、Long、BigInteger
%f	以十进制浮点数格式输出，可用于 float、double、Float、Double 或 BigDecimal
%e, %E	以科学记号浮点数格式输出，提供的数必须是 float、double、Float、Double 或 BigDecimal。%e 表示输出格式遇到字母以小写表示，如 2.13e+12，%E 表示遇到字母以大写表示
%o	以八进制整数格式输出，可用于 byte、short、int、long、Byte、Short、Integer、Long 或 BigInteger
%x, %X	以十六进制整数格式输出，可用于 byte、short、int、long、Byte、Short、Integer、Long 或 BigInteger。%x 表示字母输出以小写表示，%X 则以大写表示
%s, %S	字符串格式符号
%c, %C	以字符符号输出，提供的数必须是 byte、short、char、Byte、Short、Character 或 Integer。%c 表示字母输出以小写表示，%C 则以大写表示
%b, %B	输出 boolean 值，%b 表示输出结果会是 true 或 false，%B 表示输出结果会是 TRUE 或 FALSE。非 null 值输出是 true 或 TRUE，null 值输出是 false 或 FALSE
%h, %H	使用 Integer.toHexString(arg.hashCode()) 来得到输出结果，如果 arg 是 null，则输出 null，也常用于十六进制格式输出
%n	输出平台特定的换行符号，如果 Windows 下会置换为"\r\n"，如果是 Linux 下则会置换为'\n'，而在 Mac OS 下会置换为'\r'

提示 »» 如果想了解更多格式控制符号，可以参考：
docs.oracle.com/javase/9/docs/api/java/util/Formatter.html

从 printf() 方法的第二个自变量开始，会依次置换掉第一个自变量的格式控制符号。Byte、Short、Integer、Long、Float、Double、Character、Boolean 都是 java.lang 包下的类，在第 4 章会介绍到，这些类都是基本类型的包裹器(Wrapper)，至于 MAX_VALUE、MIN_VALUE、MIN_EXPONENT、MAX_EXPONENT、TRUE、FALSE 等，都是这些类上的静态(static)成员(其实 System 中的 out 也是)。

这个范例的输出结果如下：

```
-128 ~ 127
-32768 ~ 32767
-2147483648 ~ 2147483647
-9223372036854775808 ~ 9223372036854775807
-126 ~ 127
-1022 ~ 1023
0 ~ ffff
true ~ false
```

如果你懒得再测试一些程序片段时，还得定义类、main 程序进入点、编译、执行程序，在 JDK9 内建了 **jshell**，这是 Java 版的 REPL（Read-Eval-Print Loop）环境，也就是一个简单的、交互式的程序设计环境，可以直接使用 **jshell** 指令来进入该环境，如图 3.1 所示。

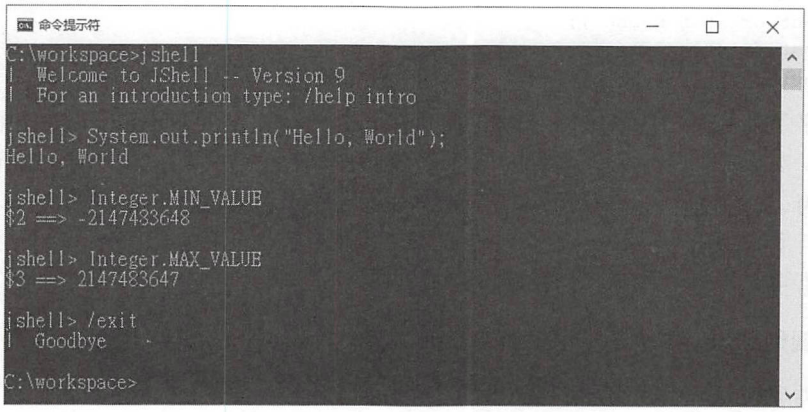


图 3.1 使用 jshell 测试小程序

在 jshell 环境中，会先执行指定的程序片段，若执行过后有传回值，会使用$n ==>的形式来显示返回值；如果想要离开 jshell 环境，可以输入/exit 或者是按下 Ctrl+D 键；若需要查看指令说明，可以输入/help，也可以在执行 jshell 时使用--help 自变量显示说明。

本书之后有一些简单的程序片段，将会适当地使用 jshell 来示范，如果看到提示文字为 jshell>开头，表示那是在 jshell 环境之中执行。

可以在输出浮点数时指定精度，例如：

```
jshell> System.out.printf("example:%.2f%n", 19.234);
example:19.23
```

也可以指定输出时至少要预留的字符宽度。例如：

```
jshell> System.out.printf("example:%6.2f%n", 19.234);
example: 19.23
```

由于预留了 6 个字符宽度，不足的部分要由空格符补上，所以执行结果会输出 example: 19.23(19.23 只占 5 个字符，所以补了一个空格在前端)。

3.1.2 变量

如果想使用基本类型数据，只要在程序中写下 10、3.14 这类数值即可。例如：

```
System.out.println(10);
System.out.println(3.14);
System.out.println(10);
```

想象一下程序中输出 10 的部分很多，如果想要一次性把它改为 20，那也要改很多地方，如果要求 JVM 有个位置可以暂存 10 这个值，每次都取出这个暂存位置的值来输出，如果将暂存位置的值改为 20，那所有输出不就都改为 20 了吗？对！这个暂存位置在程序语言中称为变量(Variable)。可以声明(Declare)变量，也就是告诉 JVM：我要有个位置。这个位置叫作×××，可以放什么类型的数据。例如：

```
int number = 10;
double PI = 3.14;
```

```
System.out.println(number);
System.out.println(PI);
System.out.println(number);
```

这个程序片段的第一行在告诉JVM，我要有个位置，这个位置叫作number，可以放int类型的数据，“=”表示指定number名称的位置会存放10，用程序术语来说的话，你声明了number变量，类型为int，使用“=”赋值运算符指定number变量的值为10。

如果之后所有输出10的部分改为20，那就只要修改number变量的指定值为20就可以了，这就是程序语言中变量的作用：用来暂存资料。

1. 基本规则

对基本类型来说，想要声明何种类型的变量，就使用byte、short、int、long、float、double、char、boolean等关键词来声明。变量在命名时有一些规则，它不可以使用数字作为开头，也不可以使用一些特殊字符，如*、&、^、%之类的字符，而变量名称不可以与Java的关键词(Keyword)同名。例如，int、float、class等就不能用来作为变量，变量名称也不可以与Java保留字(Reversed word)同名，例如goto就不能用来作为变量名称。

变量命名的风格主要以易懂为主。初学者为了方便，常使用一些简单字母来作为变量名称，这会造成日后程序维护的困难，命名变量时发生同名的情况也会增加。在Java领域中的命名习惯(Naming Convention)，通常会以小写字母开始，并在每个单词开始时第一个字母使用大写。例如：

```
int ageOfStudent;
int ageOfTeacher;
```

这种命名方式有个有趣的名称，称为驼峰式(Camel Case)命名法，可让人一眼就看出这个变量的作用。

到目前为止，本书的程序范例都是撰写在main()方法(Method)中，在方法中声明的变量称为局部变量(Local Variable)。在Java中声明一个局部变量，就会为变量配置一块内存空间，但不会给这块空间默认值，这块空间中原先可能就有无法预期的值。Java对于安全性的要求极高，不可以声明局部变量后但未指定任何值给它之前就使用变量，编译程序遇到这种情况也会编译错误，如图3.2所示。

这个程序片段声明变量score却没有指定值给它，第二行立即就要显示值，这样会出现图3.2所示的编译错误信息。

注意»» 在jshell中，声明变量时若没有使用=指定，会自动指定对应于变量类型的默认初始值。

如果在指定变量值之后，不想再改变变量值，可以在声明变量时加上**final**限定，如果后续撰写程序时，自己或别人不经意想修改final变量，就会出现编译错误，如图3.3所示。

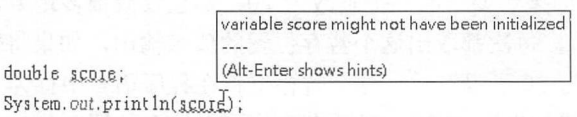


图3.2 没有初始变量就使用的编译错误

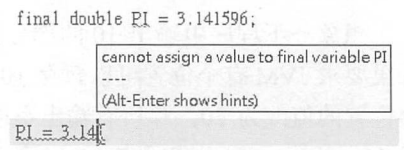


图3.3 重新指定final变量的编译错误

2. 字面常量

在 Java 中写下一个值，该值称为字面常数(Literal Constant)。在整数字面常量表示上，除了 123 这样的十进制表示法之外，还可以表示为八进制或十六进制。例如，10 这个值可以分别以十进制、十六进制及八进制表示：

```
int number1 = 12;     // 十进制表示
int number2 = 0xC;    // 十六进制表示，以 0x 开头
int number3 = 014;    // 八进制表示，以 0 开头
```

浮点数除了使用小数方式直接表示外，也可以直接使用科学记号表示。例如，以下两个变量，都是表示 0.00123 的值：

```
double number1 = 0.00123;
double number2 = 1.23e-3;
```

要表示字符的话，必须使用“'”符号括住字符。例如：

```
char size = 'S';
char lastName = '林';
```

像“'”这个符号在语法上已用来表示字符，如果只是想表示“'”这个字符呢？必须使用忽略(Escape)符号“\”，编译程序看到“\”就会忽略下一个字符，而不是将下一个字符作为程序语法的一部分。例如，要表示“'”就要用“\'”：

```
char symbol = '\'';
```

表 3.2 所示是常用的一些忽略符号。

表 3.2 常用忽略符号

忽略符号	说 明
\\	反斜杠\
\'	单引号'
\"	双引号"
\uxxxx	以十六进制数指定 Unicode 字符输出，x 表示数字
\xxx	以八进制数指定 Unicode 字符输出，x 表示数字
\b	倒退一个字符
\f	换页
\n	换行
\r	光标移至行首

例如，要使用 Unicode 字符编码(Code Point)来输出 Hello，代码如下：

```
jshell> System.out.println("\u0048\u0065\u006C\u006C\u006F");
Hello
```

提示 »» \uxxxx 表示法，可用在想使用的字符无法以打字方式输入的情况。事实上，如果在程序中输入了汉字字符，编译时也会展开为\uxxxx 表示法，这些细节在第 4 章关于字符串的内容中介绍。

boolean 类型可指定的值只有 true 与 false。例如：

```
boolean flag = true;
boolean condition = false;
```

3. 数字常量表示法

从 Java SE 7 开始，撰写整数或浮点数常量时可以使用下划线更清楚地表示某些数字。例如：

```
int number1 = 1234_5678;
double number2 = 3.141_592_653;
```

有时候，想要以二进制方式表示某个值，则可以用 0b 作为开头。例如：

```
int mask = 0b101010101010;    // 用二进制表示十进制整数 2730
```

上面的程序片段也可以结合下划线，这样就更清楚了：

```
int mask = 0b1010_1010_1010;    // 用二进制表示十进制整数 2730
```

3.1.3 运算符

程序的目的简单地说就是运算，除了运算还是运算，在程序语言中提供运算功能的就是运算符(Operator)。

1. 算术运算

与算术相关的运算符有+、-、*、/，也就是加、减、乘、除这类运算符，另外%称为模数运算符或余除运算符。算术运算符使用上与学过的加减乘除一样，也是先乘除后加减。例如程序代码 1 + 2 * 3 的结果会是 7，而 2 + 2 + 8 / 4 结果会是 6：

```
jshell> 1 + 2 * 3;
$1 ==> 7
jshell> 2 + 2 + 8 / 4;
$2 ==> 6
```

如果想要的是 2 + 2 + 8 相加后，再除以 4，则需加上括号表示运算先后顺序。例如，以下程序代码显示的是 3：

```
jshell> (2 + 2 + 8) / 4;
$3 ==> 3
```

%运算符计算的结果是除法后的余数，例如 10 % 3 会得到余数 1。一个使用%的例子是数字循环，假设有个立方体要进行 360° 旋转，每次要在角度上加 1，而 360° 后必须复归为 0 重新计数，这时可以这么撰写：

```
int count = 0;
...
count = (count + 1) % 360;
```

提示 »» 可在运算符的两边各留一个空格，这样比较容易阅读。

2. 比较、条件运算

数学上有大于、等于、小于的比较运算，Java 中也提供了这些运算符，这些运算符称为比较运算符(Comparison Operator)，它们有大于(>)、不小于(>=)、小于(<)、不大于(<=)、等于(==)

以及不等于(!=)，比较条件成立时以 boolean 类型 true 表示，比较条件不成立时以 false 表示。以下程序片段示范了几个比较运算的使用：

Basic_Comparison.java

```
package cc.openhome;

public class Comparison {
    public static void main(String[] args) {
        System.out.printf("10 >  5 结果 %b%n", 10 > 5);
        System.out.printf("10 >= 5 结果 %b%n", 10 >= 5);
        System.out.printf("10 <  5 结果 %b%n", 10 < 5);
        System.out.printf("10 <= 5 结果 %b%n", 10 <= 5);
        System.out.printf("10 == 5 结果 %b%n", 10 == 5);
        System.out.printf("10 != 5 结果 %b%n", 10 != 5);
    }
}
```

程序的执行结果如下所示：

```
10 >  5 结果 true
10 >= 5 结果 true
10 <  5 结果 false
10 <= 5 结果 false
10 == 5 结果 false
10 != 5 结果 true
```

注意 »» ==是由两个连续的=组成，而不是一个=，一个=是指定运算，这一点必须特别注意。例如，变量 x 与 y 要比较是否相等，应该是写成 x == y，而不是写成 x = y，后者作用是将 y 的值指定给 x，而不是比较 x 与 y 是否相等。

提示 »» 对于类类型声明的参考名称来说，两个参考名称使用==比较时，是比较两个名称是否参考至同一对象，在第 4 章介绍对象时会再详细介绍。

Java 有个条件运算符(Conditional Operator)，使用方式如下：

```
条件式 ? 成立返回值 : 失败返回值
```

条件运算符返回值根据条件式结果而定，如果条件式结果为 true，则返回“:”前的值，若为 false，则返回“:”后的值。例如，若 score 是 int 声明，储存了使用者输入的学生成绩，以下程序片段可用来判断学生是否及格：

```
System.out.printf("该生是否及格?%c%n", score >= 60 ? '是' : '否');
```

条件运算符使用适当的话可以少写几句程序代码。例如，若 number 是 int 声明，储存用户输入的数字，则以下程序片段可判断奇数或偶数：

```
System.out.printf("是否为偶数?%c%n", (number % 2 == 0) ? '是' : '否');
```

同样的程序片段，若改用本章稍后要介绍的 if...else 语法，则要如下撰写：

```
if(number % 2 == 0) {
    System.out.println("是否为偶数?是");
}
else {
    System.out.println("是否为偶数?否");
}
```

3. 逻辑运算

在逻辑上有所谓的“且”(AND)、“或”(OR)与“反相”(NOT)，在 Java 中也提供对应的逻辑运算符(Logical Operator)，分别为&&(AND)、||(OR)及!(NOT)。看看以下的程序片段会输出什么结果？

```
jshell> int number = 75;
number ==> 75

jshell> number > 70 && number < 80;
$2 ==> true

jshell> number > 80 || number < 75;
$3 ==> false

jshell> !(number > 80 || number < 75);
$4 ==> true
```

三段描述句分别输出 true、false 与 true 三种结果，分别表示 number 大于 70 且小于 80 为真、number 大于 80 或小于 75 为假、number 大于 80 或小于 75 的反相为真。

&&与||有所谓快捷方式运算(Short-Circuit Evaluation)。因为 AND 只要其中一个为假，就可以判定结果为假，所以对&&来说，只要左操作数(Operand)评估为 false，就会直接返回 false，不会再去运算右操作数。因为 OR 只要其中一个为真，就可以判定结果为真，所以对||来说，只要左操作数评估为 true，就会直接返回 true，就不会再去运算右操作数。

来举个运用快捷方式运算的例子。在 Java 中两个整数相除，若除数为 0 会发生 ArithmeticException，代表除 0 的错误。以下运用&&快捷方式运算避免了这个问题：

```
if(b != 0 && a / b > 5) {
    // 做一些事...
}
```

在这个程序片段中，变量 a 与 b 都是 int 类型，如果 b 为 0，&&左边操作数结果就是 false，直接判断整个&&的结果应是 false，不用再去评估右操作数，从而避免了 a / b 而 b 等于 0 时的除零错误。

4. 位运算

在 Java 中的位运算符(Bitwise Operator)，分别是移位、&(AND)、|(OR)、^(XOR)与~(非)。如果不会基本位运算，可以从以下范例了解各个位运算结果：

Basic_Bitwise.java

```
package cc.openhome;
public class Bitwise {
    public static void main(String[] args) {
        System.out.println("AND 运算：");
        System.out.printf("0 AND 0 %5d%n", 0 & 1);
        System.out.printf("0 AND 1 %5d%n", 0 & 1);
        System.out.printf("1 AND 0 %5d%n", 1 & 0);
        System.out.printf("1 AND 1 %5d%n", 1 & 1);

        System.out.println("\nOR 运算：");
        System.out.printf("0 OR 0 %6d%n", 0 | 0);
```

```
        System.out.printf("0 OR 1 %6d%n", 0 | 1);
        System.out.printf("1 OR 0 %6d%n", 1 | 0);
        System.out.printf("1 OR 1 %6d%n", 1 | 1);

        System.out.println("\nXOR 运算: ");
        System.out.printf("0 XOR 0 %5d%n", 0 ^ 0);
        System.out.printf("0 XOR 1 %5d%n", 0 ^ 1);
        System.out.printf("1 XOR 0 %5d%n", 1 ^ 0);
        System.out.printf("1 XOR 1 %5d%n", 1 ^ 1);
    }
}
```

执行结果就是各个位运算的结果：

```
AND 运算:
0 AND 0      0
0 AND 1      0
1 AND 0      0
1 AND 1      1

OR 运算:
0 OR 0       0
0 OR 1       1
1 OR 0       1
1 OR 1       1

XOR 运算:
0 XOR 0      0
0 XOR 1      1
1 XOR 0      1
1 XOR 1      0
```

位运算是逐位运算。例如，10010001 与 01000001 做 AND 运算，是一个一个位对应运算，答案就是 00000001。补码运算是将所有位 0 变 1，1 变 0。例如，00000001 经补码运算就会变为 11111110。例如：

```
jshell> byte number = 0;
number ==> 0

jshell> ~number;
$2 ==> -1
```

上面的程序片段会显示-1，因为 `byte` 占内存 1 字节，`number` 储存的 0 在内存中是位 00000000，经补码运算就变成 11111111，这个数在计算机中用整数表示则是-1。

注意>>> 逻辑运算符与位运算符也是经常被混淆的，像是&&与&、||与|，初学时可得多注意。

在位运算上，Java 移位运算有左移(<<)与右移(>>)两个运算符。左移运算符会将所有位往左移指定位数，左边被挤出去的位会被丢弃，而右边补上 0；右移运算符则相反，会将所有位往右移指定位数，右边被挤出去的位会被丢弃，至于最左边补上原来的位，如果左边原来是 0 就补 0，是 1 就补 1。还有个>>>运算符，这个运算符在右移后，最左边一定是补 0。

使用左移运算来做简单的 2 次方运算示范：

Basic_Shift.java

```
package cc.openhome;

public class Shift {
    public static void main(String[] args) {
        int number = 1;
        System.out.printf( "2 的 0 次方: %d%n", number);
        System.out.printf( "2 的 1 次方: %d%n", number = number << 1);
        System.out.printf( "2 的 2 次方: %d%n", number = number << 2);
        System.out.printf( "2 的 3 次方: %d%n", number = number << 3);
    }
}
```

执行结果：

```
2 的 0 次方: 1
2 的 1 次方: 2
2 的 2 次方: 4
2 的 3 次方: 8
```

实际来左移看看就知道为何可以这样做次方运算了：

00000001 → 1

00000010 → 2

00000100 → 4

00001000 → 8

提示 »» 位运算平时并不常用，通常应用于图像处理、文字编码等场合。例如，以下网址(《乱码1/2》)就有一些位运算的例子：
http://openhome.cc/Gossip/Encoding/

5. 递增、递减运算

在程序中对变量递增 1 或递减 1 是很常见的运算。例如：

```
int i = 0;
i = i + 1;
System.out.println(i);
i = i - 1;
System.out.println(i);
```

这个程序片段会分别显示出 1 与 0 两个数，可以使用递增、递减运算符来撰写程序：

```
int i = 0;

i++;
System.out.println(i);
i--;
System.out.println(i);
```

那么哪个写法比较好呢？就简洁度而言，使用++、--的写法比较好，就效率而言，其实没有差别。因为如果你写 `i = i + 1`，编译程序会自动帮你改成 `i++`，同样地，如果你写 `i = i - 1`，编译程序会自动帮你改为 `i--`。

上面的程序片段还可以再简洁一些：

```
int i = 0;
System.out.println(++i);
System.out.println(--i);
```

可以将++或--运算符撰写在变量的前或后，不过两种写法有差别，将++或--运算符写在变量前，表示先将变量值加或减 1，然后再返回变量值；将++或--运算符写在变量后，表示先返回变量值，然后再对变量加或减 1。例如：

```
int i = 0;
int number = 0;
number = ++i;    // 结果相当于 i = i + 1; number = i;
System.out.println(number);
number = --i;    // 结果相当于 i = i - 1; number = i;
System.out.println(number);
```

在这个程序片段中，number 的值会前后分别显示为 1 与 0。再来看个例子：

```
int i = 0;
int number = 0;
number = i++;    // 相当于 number = i; i = i + 1;
System.out.println(number);
number = i--;    // 相当于 number = i; i = i - 1;
System.out.println(number);
```

在这个程序片段中，number 的值会前后分别显示为 0 与 1。

6. 指定运算

到目前为止我们仅用过一个指定运算符，也就是=这个运算符。事实上，指定运算符还有表 3.3 所示的一些。

表 3.3 指定运算符

指定运算符	范 例	结 果
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b

3.1.4 类型转换

类型与变量看似简单，但每种程序语言可能都有其不同的细节。接下来要介绍的类型转换概念，看似只是认证题目中很常考，然而实务上，确实也有些情况，因为忽略了类型转换而误踏地雷的例子，因此还是要有所了解。

首先，如果写了以下程序片段：

```
double PI = 3.14;
```

这个片段编译时没有问题，但如果你写了图 3.4 所示的程序片段，为什么得到了 possible loss of precision

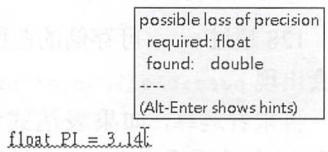


图 3.4 遗失精度

的编译错误？这是因为在程序中写下一个浮点数时，编译程序默认会使用 **double** 类型。就图 3.4 来说，如果想要将 double 长度的数据指定给 float 类型变量，编译程序就会很贴心地提醒 double 类型放到 float 变量，会因为 8 字节数据要放到 4 字节空间，而遗失 4 字节的数据。

有两种方式可以避免这个错误。第一种方式是在 3.14 后面加上 **F**，这会告诉编译程序，请用 float 来储存 3.14 这个值。例如：

```
float PI = 3.14F;
```

另一种方式是明确告诉编译程序，就是要将 double 类型的 3.14 丢(Cast)到 float 变量中，请编译程序住嘴：

```
float PI = (float) 3.14;
```

编译程序看到 double 类型的 3.14 要指定给 float 变量，本来会告诉你将遗失精度，但你使用 (float) 语法告诉编译程序，你就是要将 double 类型的 3.14 指定给 float 变量，别再啰唆了，于是编译程序就住嘴不讲话了，于是编译通过。既然你不要编译程序啰唆了，那执行时候出错，后果请自负，也就是如果真的因为遗失精度而发生程序错误，那绝不是编译程序的问题。

再来看整数的部分。如果写下：

```
int number = 10;
```

这没有问题。如果写下图 3.5 所示代码，编译时会得到 integer number too large 的错误。

也许你以为原因是 int 变量 number 装不下 2 147 483 648，因为 int 类型最大值是 2 147 483 647，认为这样可以解决问题，如图 3.6 所示。

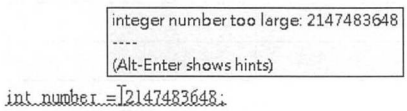


图 3.5　整数太大

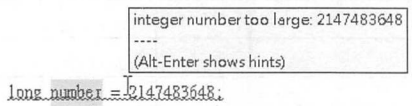


图 3.6　还是整数太大

事实上，并非是 number 装不下 2 147 483 648(如果是的话，编译错误信息应该是 possible loss of precision)，而是程序中写下一个整数时，默认是使用不超过 int 类型长度。2 147 483 648 超出了 int 类型的长度，就要直接告诉编译程序，用 long 来配置整数的长度，也就是在数字后加上个 **L**：

```
long number = 2147483648L;
```

这样就可以通过编译了。在程序中写下一个整数时，默认是使用不超过 int 类型的长度，所以下面的程序可以通过编译：

```
byte number = 10;
```

因为 10 是在 byte 可储存的范围中，不过这样不行：

```
byte number = 128;
```

128 超过 byte 可存储的范围，于是会使用 int 存储 128，若要将 int 类型存储至 byte 变量，就会出现 possible loss of precision 的编译错误。

再来看运算，如果表达式中包括不同类型数值，则运算时以长度最长的类型为主，其他数值自动提升(Promote)类型。例如：

```
int a = 10;
double b = a * 3.14;
```

在这个程序片段中，a 是 int 类型，而写下的 3.14 默认是 double，所以 a 的值被提至 double 空间进行运算。

如果操作数都是不大于 **int** 的整数，则自动地全部提升为 **int** 类型进行运算。图 3.7 所示代码片段通不过编译。

虽然 a 与 b 都是 short 类型，但 Java 在运算整数时，如果全部的操作数都是不大于 int，那么一律在 int 的空间中运算，int 的运算结果要放到 short，编译程序就又会提醒遗失精度的问题。所以你要告诉编译程序，就是要将 int 的运算结果丢到 short，请它住嘴：

```
short a = 1;
short b = 2;
short c = (short) (a + b);
```

类似地，图 3.8 所示的程序片段通不过编译。

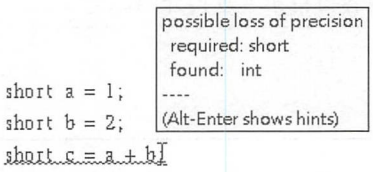


图 3.7 又遗失精度了

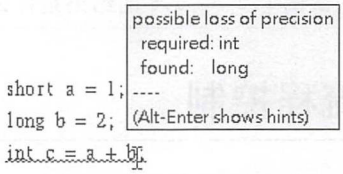


图 3.8 这次怎么又遗失精度

如果表达式中包括不同类型，则运算时会以最长的类型为主。以图 3.8 来说，b 是 long 类型，所以 a 也被提至 long 空间中做运算，long 的运算结果要放到 int 变量 c，那么就会被编译程序啰唆精度遗失了。如果这真的是你想要的，那就叫编译程序住嘴吧！

```
short a = 1;
long b = 2;
int c = (int) (a + b);
```

那么以下程序你觉得会显示多少？

```
System.out.println(10 / 3);
```

答案是 3，而不是 3.333333……，因为 10 与 3 会在 int 长度的空间中做运算，因此不会做浮点数表示，如果想得到 3.333333……的结果，那么必须有一个操作数是浮点数。例如：

```
System.out.println(10.0 / 3);
```

很无聊对吧！好像只是在玩弄语法似的。那么，看看下面的程序片段有没有问题：

```
int count = 0;
while(someCondition) {
    if(count + 1 > Integer.MAX_VALUE) {
        count = 0;
    }
    else {
        count++;
    }
    ...
}
```

这个程序片段想做的是，在某些情况下不断递增 count 的值，如果 count 超过上限就归零，

在这里以 int 类型的最大值为上限。程序逻辑看似没错，但 count + 1 > Integer.MAX_VALUE 永远不会是 true，如果 count 已经到了 2 147 483 647，也就是 int 的最大值，此时内存中的字节会是：

```
01111111 11111111 11111111 11111111
```

count + 1 则会变为：

```
11111111 11111111 11111111 11111111
```

字节第一个位是 1，在 Java 中表示一个负数，上例也就是表示-2 147 483 648。简单来讲，最后 count + 1 会因为超出了 int 可储存范围而溢值，count + 1 > Integer.MAX_VALUE 永远不会成立。

提示 »» 以下网址(《Promotion 与 Cast》)还有两个例子，你可以想想看问题是什么：
http://openhome.cc/Gossip/JavaEssence/PromotionCast.html
有个例子跟类型转换无关，但却是很重要的概念：不要对浮点数做相等性运算。你可以先试试 1.0 - 0.8 的结果是什么，详细内容留待第 4 章再介绍。

3.2 流程控制

现实生活中待解决的事千奇百怪，在计算机发明之后，想要使用计算机解决的需求也是各式各样："如果"发生了……，就要……；"对于"……，就一直执行……；"如果"……，就"中断"……。为了告诉计算机特定条件下该执行的动作，要使用各种条件式来定义程序执行的流程。

3.2.1 if...else 条件式

为了应付"如果×××成立"就要……，"否则"就要……的需求，Java 提供了 if...else 条件式。语法如下：

```
if(条件式) {
    描述句;
}
else {
    描述句;
}
```

条件式运算结果为 true 会执行 if 的{ }中的描述句，否则执行 else 的{ }中的描述句。如果条件式不成立并不想做任何事，则 else 可以省略。

下面是运用 if...else 判断数字为奇数或偶数的范例：

Basic Odd.java

```
package cc.openhome;

public class Odd {
    public static void main(String[] args) {
        int input = 10;
        int remain = input % 2;
        if(remain == 1) { // 余数为 1 就是奇数
```

```
            System.out.printf("%d 为奇数%n", input);
        }
        else {
            System.out.printf("%d 为偶数%n", input);
        }
    }
}
```

提示 »» 范例中的 input 变量，实际上可由使用者输入取得值。如何让使用者输入，第 4 章会介绍。

如果 if 或 else 中只有一行描述句，则{与}可以省略。不过为了可读性与可维护性而言，现在建议是就算只有一行描述句，也要撰写{与}明确定义范围。

提示 »» Apple 曾经提交一个 iOS 上的安全更新：http://support.apple.com/kb/HT6147。原因是在某个函式中有两个连续的缩排：

```
...
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
...
```

因为缩排在同一层，阅读程序代码时大概也就没注意到，又没有{}定义区块，结果就是 goto fail，无论如何都会被执行到的错误。

某些人会撰写所谓的 if...else if 语法：

```
if(条件式一) {
    ...
}
else if(条件式二) {
    ...
}
else {
    ...
}
```

其实 Java 中并没有真的 if...else if 语法，这是省略{}加上程序代码排版后的结果，如果不省略{}，原本的程序应该是：

```
if(条件式一) {
    ...
}
else {
    if(条件式二) {
        ...
    }
    else {
        ...
    }
}
```

如果条件式一不满足，就执行 else 中的语句，而在这里进行条件式二测试，如果满足就执行条件式二{}中的语句。如果省略了第一个 else 的{}：

```
if(条件式一) {
    ...
```

```
    }
    else
        if(条件式二) {
            ...
        }
        else {
            ...
        }
```

由于 Java 是个自由格式语言，可以适当地排列程序代码段，就会变成刚才看到的 `if...else if` 写法，这样可读性反而好一些。例如，应用在处理学生的成绩等级问题上。

Basic Level.java

```
package cc.openhome;

    public class Level {
        public static void main(String[] args) {
            int score = 88;
            char level;
            if(score >= 90) {
                level = 'A';
            }
            else if(score >= 80 && score < 90) {
                level = 'B';
            }
            else if(score >= 70 && score < 80) {
                level = 'C';
            }
            else if(score >= 60 && score < 70) {
                level = 'D';
            }
            else {
                level = 'E';
            }
            System.out.printf("得分等级：%c%n", level);
        }
}
```

不过就这个例子而言，效率并不好，在最差的情况下，也就是成绩小于 60 分的情况下，总共要从 `score` 变量中取出 7 次的值。就这个例子而言，改用接下来要说明的 `switch` 会比较好。

提示>>> 如果 `isOpened` 是 `boolean` 类型，想在 `if` 中做判断，请别这么写：

```
if(isOpened == true) {
    ...
}
```

这样程序也可以正确执行，只不过很逊色。你只要这么写就好了：

```
if(isOpened) {
    ...
}
```

3.2.2 switch 条件式

在 JDK7 前，`switch` 可用于比较整数、字符、`Enum`，从 JDK7 开始，增加了对字符串的比较。`Enum` 与 `switch` 后面会再详细说明。`switch` 的语法架构如下：

```
switch(变量或表达式) {
    case 整数、字符、字符串或Enum:
        描述句;
        break;
    case 整数、字符、字符串或Enum:
        描述句;
        break;
    ...
    default:
        描述句;
}
```

首先看看 switch 的括号，当中放置要取得值的变量或表达式，值必须是整数、字符、字符串或 Enum，取得值后会开始与 case 中设定的整数、字符、字符串或 Enum 比较，如果符合就执行之后的描述句，直到遇到 break 离开 switch 区块；如果没有符合的整数、字符、字符串或 Enum，则会执行 default 后的描述句。default 不一定需要，如果没有默认要处理的动作，可以省略 default。

来看看前面范例的 Level 类，如何改用 switch 运行：

Basic Level2.java

```
package cc.openhome;

public class Level2 {
    public static void main(String[] args) {
        int score = 88;
        int quotient = score / 10;
        char level;
        switch(quotient) {
            case 10:
            case 9:
                level = 'A';
                break;
            case 8:
                level = 'B';
                break;
            case 7:
                level = 'C';
                break;
            case 6:
                level = 'D';
                break;
            default:
                level = 'E';
        }
        System.out.printf("得分等级：%c%n", level);
    }
}
```

在这个程序中，使用除法并取得运算后的商数，如果大于 90，除以 10 的商数一定是 9 或 10(100 分时)，在 case 10 中没有任何的描述，也没有使用 break，所以继续往下执行，直到遇到 break 离开 switch 为止，所以学生成绩 100 分的话，会显示 A 的成绩等级；如果比较条件不是 10 到 6 这些值，则会执行 default 下的描述，这表示商数小于 6，所以学生成绩等级就显示为 E 了。

这个程序与前一个范例使用 if...else 来判断成绩等级有何不同？这个程序只会在一开始的 switch 括号中取出 quotient 变量的值一次，然后将这个值与之后 case 比较，所以效率比较高。

3.2.3 for 循环

在 Java 中如果要进行重复性指令执行，可以使用 **for** 循环。基本语法之一如下：

```
for(初始式; 执行结果必须是 boolean 的重复式; 重复式) {
    描述句;
}
```

for 循环语法的圆括号中，初始式只执行一次，所以通常用来声明或初始变量，如果是声明变量，结束 for 循环后变量就会消失。第一个分号后，则是每次执行循环体前会执行一次，且必须是 true 或 false 的结果，true 就会执行循环体，false 就会结束循环。第二个分号后，则是每次执行完循环体后会执行一次。

实际来看 for 循环范例，在 jshell 中显示 1 到 10：

```
jshell> for(int i = 1; i <= 10; i++) {
   ...>     System.out.println(i);
   ...> }
1
2
3
4
5
6
7
8
9
10
```

这个程序直接读来，就是从 i 等于 1，只要 i 小于等于 10 就执行循环体(显示 i)，然后递增 i，这是 for 循环常见的应用方式。如果 for 本体只有一行描述句，则{}可以省略，不过为了可读性与可维护性而言，现在建议是就算只有一行描述句，也要撰写{}明确定义范围。

在介绍 for 循环时，许多书籍或文件很喜欢用的范例就是显示九九表。下面就用这个例子来示范：

Basic NineNineTable.java

```
package cc.openhome;

public class NineNineTable {
    public static void main(String[] args) {
        for(int j = 1; j < 10; j++) {
            for(int i = 2; i < 10; i++) {
                System.out.printf("%d*%d=%2d ",i, j,  i * j);
            }
            System.out.println();
        }
    }
}
```

执行结果如下：

```
2*1= 2 3*1= 3 4*1= 4 5*1= 5 6*1= 6 7*1= 7 8*1= 8 9*1= 9
2*2= 4 3*2= 6 4*2= 8 5*2=10 6*2=12 7*2=14 8*2=16 9*2=18
2*3= 6 3*3= 9 4*3=12 5*3=15 6*3=18 7*3=21 8*3=24 9*3=27
2*4= 8 3*4=12 4*4=16 5*4=20 6*4=24 7*4=28 8*4=32 9*4=36
```

```
2*5=10 3*5=15 4*5=20 5*5=25 6*5=30 7*5=35 8*5=40 9*5=45
2*6=12 3*6=18 4*6=24 5*6=30 6*6=36 7*6=42 8*6=48 9*6=54
2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49 8*7=56 9*7=63
2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64 9*8=72
2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```

事实上，for 循环语法只是将 3 个复合描述区块写在圆括号中而已，第一个描述区块只会执行一次，第二个描述区块专门判断是否继续下一个循环，而第三个描述区块只是一般的描述句。

for 圆括号中的每个描述区块是以分号“;”作分隔，而在一个描述区块中若想写两个以上的描述句，则使用逗号“,”作分隔。如果有兴趣的话，研究一下下面九九表的写法，只使用了一个 for 循环就可以完成九九表打印，执行结果与上一个范例相同，但就可读性而言，并不建议这么写。

Basic NineNineTable2.java

```
package cc.openhome;
    public class NineNineTable2 {
        public static void main(String[] args) {
            for (int i = 2, j = 1;  j < 10; i = (i==9)?((++j/j)+1):(i+1)) {
                System.out.printf("%d*%d=%2d%c", i, j,  i * j, (i==9 ? '\n' : ' '));
            }
        }
}
```

提示>>> for 循环圆括号中第二个复合描述区块若没有撰写，默认就是 true。所以，如果看到有人如下撰写的话，表示无穷循环：

```
for(;;) {
    ...
}
```

3.2.4 while 循环

Java 提供 while 循环，可根据指定条件式来判断是否执行循环体。语法如下所示：

```
while(条件式) {
    描述句;
}
```

如果循环体只有一个描述句，则 while 的{}可以省略不写，但为了可读性，建议还是撰写。while 主要用于停止条件必须在执行时期判断的重复性动作。例如，在一个用户输入接口，用户可能输入的学生名称个数未知，只知道要结束时会输入 quit，就可以使用 while 循环。

下面有个游戏，看谁可以在更长时间内不撞到 5 这个数字。

Basic RandomStop.java

```
package cc.openhome;

    public class RandomStop {
        public static void main(String[] args) {
            while(true) {  ← ❶ 直接执行循环
                int number = (int) (Math.random() * 10);  ← ❷ 随机产生 0~9 的数
                System.out.println(number);
```

```
                if(number == 5) {
                    System.out.println("I hit 5...Orz");
                    break; ◄── ❸ 如果遇到 5 就离开循环
                }
            }
        }
}
```

这个范例的 while 判断式直接设为 true❶，表示每次 while 重来就直接执行循环体，Math.random()会随机产生 0.0～1.0 的值，乘上 10 再裁掉小数部分，表示产生 0~9 的数❷，在 while 循环中如果执行到 break，会离开循环体了❸。

一个参考的执行结果如下：

```
9
5
I hit 5...Orz
```

while 循环有时称为前测试循环，因为会在循环执行前就进行条件判断。如果想先执行一些动作，再判断要不要重复，则可以使用 do...while，又称为后测试循环。do...while 的语法如下所示：

```
do {
    描述句;
} while(条件式);
```

注意，while 后面是以分号“;”作为结束，这个经常被忽略。将上一个范例改为 do...while，可以少写一个 boolean 判断：

Basic RandomStop2.java

```
package cc.openhome;

public class RandomStop2 {
    public static void main(String[] args) {
        int number;
        do {
            number = (int) (Math.random() * 10); ◄── ❶ 先随机产生 0~9 的数
            System.out.println(number);
        } while(number != 5); ◄── ❷ 再判断要不要重复执行
        System.out.println("I hit 5...Orz");
    }
}
```

RandomStop 有 while 与 if 两个需要 boolean 判断的地方，这主要是因为一开始 number 的值并没有产生，所以只好先进入 while 循环。使用 do...while，先产生 number❶，再判断要不要执行循环❷，刚好可以解决这个问题。

3.2.5 break、continue

break 可以离开当前 switch、for、while、do...while 的区块，并执行区块后下一个描述句，在 switch 中主要用来中断下一个 case 比较，在 for、while 与 do...while 中，主要用于中断当前循环。

continue 的作用与 break 类似，不过使用于循环，break 会结束区块执行，而 continue 只会

略过之后描述句，并回到循环区块开头进行下一次循环，而不是离开循环。例如：

```
jshell> for(int i = 1; i < 10; i++) {
   ...>     if(i == 5) {
   ...>         break;
   ...>     }
   ...>     System.out.printf("i = %d%n", i);
   ...> }
i = 1
i = 2
i = 3
i = 4
```

这段程序会显示 i = 1 到 i = 4，因为在 i 等于 5 时就会执行 break 而离开循环。再看下面这个程序：

```
jshell> for(int i = 1; i < 10; i++) {
   ...>     if(i == 5) {
   ...>         continue;
   ...>     }
   ...>     System.out.printf("i = %d%n", i);
   ...> }
i = 1
i = 2
i = 3
i = 4
i = 6
i = 7
i = 8
i = 9
```

这段程序会显示 i =1 到 i =4，以及 i =6 到 i =9。当 i=5 时，会执行 continue 直接略过之后描述句，也就是该次的 System.out.printf()行并没有被执行，直接从区块开头执行下一次循环，所以 i = 5 没有被显示。

break 与 continue 还可以配合标签使用。例如，本来 break 只会离开 for 循环，设定标签与区块，则可以离开整个区块。

```
jshell> back : {
   ...>     for(int i = 0; i < 10; i++) {
   ...>         if(i == 9) {
   ...>             System.out.println("break");
   ...>             break back;
   ...>         }
   ...>     }
   ...>     System.out.println("test");
   ...> }
break
```

程序执行结果会显示 break。back 是个标签，当 break back;时，返回至 back 标签处，之后整个 back 区块不执行而跳过，所以 System.out.println("test")行不会被执行。

continue 也有类似的用法，只不过标签只能设定在 for 之前。例如：

```
jshell> back1:
   ...> for(int i = 0; i < 10; i++){
   ...>     back2:
   ...>     for(int j = 0; j < 10; j++) {
```

```
...>          if(j == 9) {
...>              continue back1;
...>          }
...>      }
...>      System.out.println("test");
...> }

jshell>
```

continue配合标签，可以自由地跳至任何一层for循环，可以试试continue back1与continue back2的不同，设定back1时，System.out.println("test")不会被执行。

3.3 重点复习

在Java中的基本类型主要可区分为整数、字节、浮点数、字符与布尔。整数可细分为short整数(占2字节)、int整数(占4字节)与long整数(占8字节)。byte类型顾名思义，长度就是1字节。浮点数可分为float浮点数(占4字节)与double浮点数(占8字节)。char类型用来存储'A'、'B'、'林'等字符符号。在JDK9中，Java的字符采用Unicode 8.0编码，JVM成果采用UTF-16 Big Endian，所以每个字符数据类型占2字节。boolean类型可表示true与false。如果存储值超出类型范围，称为溢值，会造成程序不可预期的结果。

在程序中看到//符号，这是Java程序中的单行批注，另一个批注符号是/*与*/包括的多行批注。

数据暂存位置在程序语言中称为变量。对基本类型来说，想要声明何种类型的变量，就使用byte、short、int、long、float、double、char、boolean等关键词来声明。变量在命名时有一些规则，它不可以使用数字作为开头，也不可以使用一些特殊字符，如*、&、^、%之类的字符，而变量名称不可以与Java关键词同名，也不可以与Java保留字同名。

在Java领域中的命名惯例，通常会以小写字母开始，并在每个单词开始时第一个字母使用大写，称为驼峰式命名法。

在方法中声明的变量称为局部变量，不可以声明局部变量后但未指定任何值给它之前就使用变量，编译程序遇到这种情况的也会编译错误。在声明变量时加上final限定，如果后续撰写程序时，自己或别人想修改final变量，就会出现编译错误。

从在Java SE 7开始，撰写整数或浮点数字面常量时可以使用下划线以更清楚地表示某些数字。

==是由两个连续的=组成，而不是一个=，一个=是指定运算，这一点必须特别注意。例如，若变量x与y要比较是否相等，应该是写成x == y，而不是写成x = y，后者作用是将y的值指定给x，而不是比较x与y是否相等。

&&与||有所谓快捷方式运算。因为AND只要其中一个为假，就可以判定结果为假，所以对&&来说，只要左操作数评估为false，就会直接返回false，不会再去运算右操作数。因为OR只要其中一个为真，就可以判定结果为真，所以对||来说，只要左操作数评估为true，就会直接返回true，就不会再去运算右操作数。

将++或--运算符写在变量前，表示先将变量值加或减1，然后再返回变量值；将++或--运算符写在变量后，表示先返回变量值，然后再对变量加或减1。

在程序中写下一个浮点数时，编译程序默认会使用 double 类型。程序中写下一个整数时，默认不超过 int 类型长度。如果表达式中包括不同类型数值，则运算时以长度最长的类型为主，其他数值自动提升类型。如果操作数都是不大于 int 的整数，则自动地全部提升为 int 类型进行运算。

在 JDK7 之后，switch 可用于比较整数、字符、字符串与 Enum。

for 循环语法的圆括号中，初始式只执行一次，所以通常用来声明或初始变量，如果是声明变量，结束 for 循环后变量就会消失。第一个分号后，则是每次执行循环体前会执行一次，且必须是 true 或 false 的结果，true 就会执行循环体，false 就会结束循环。第二个分号后，则是每次执行完循环体后会执行一次。for 圆括号中的每个描述区块是以分号“;”作分隔，而在一个描述区块中若想写两个以上的描述句，则使用逗号“,”作分隔。

3.4 课后练习

1. 如果有 m 与 n 两个 int 变量，分别储存 1000 与 495 两个值，请使用程序算出最大公因子。

2. 在三位的整数中，例如 153 可以满足 $1^3 + 5^3 + 3^3 = 153$，这样的数称为阿姆斯特朗(Armstrong)数，试以程序找出所有三位数的阿姆斯特朗数。

Chapter 4 认识对象

学习目标

- 区分基本类型与类类型
- 了解对象与参考的关系
- 从打包器认识对象
- 以对象观点看待数组
- 认识字符串的特性
- 学会查询 API 文件

严禁网络传播本PDF，违者责任自负！

4.1 类与对象

Java 中有基本类型与类类型两个类型系统，第 3 章介绍过基本类型，本章将介绍类类型。使用 Java 撰写程序几乎都在使用对象(Object)，要产生对象必须先定义类(Class)，类是对象的设计图，对象是类的实例(Instance)。

不卖弄术语了，让我们开始吧！

4.1.1 定义类

正式开始说明如何使用 Java 定义类之前，先来看看，设计衣服是如何进行的。先要有个衣服的设计图，上头定义了衣服的款式与颜色、尺寸，然后根据设计图制作出实际的衣服，每件衣服都是同一款式，但会拥有自己的颜色与尺寸，每个衣服都会别上一个名牌，这个名牌只能别在同款式的衣服上，如图 4.1 所示。

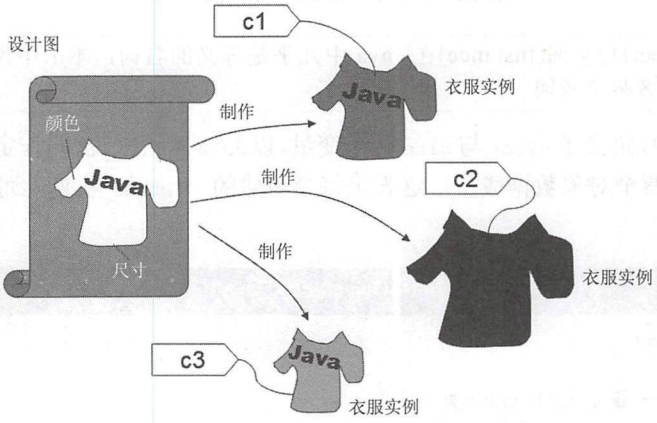


图 4.1 设计图、制作、实例与款式名牌

如果今天要设计的就是有关服饰设计的软件，那如何使用 Java 撰写呢？先在程序中定义类，这相当于图 4.1 中衣服的设计图：

```
class Clothes {
    String color;
    char size;
}
```

类定义时使用 **class** 关键词，名称使用 `Clothes`，相当于为衣服设计图取名为 Clothes，衣服的颜色用字符串表示，也就是 `color` 变量，可储存`"red"`、`"black"`、`"blue"`等值，衣服的尺寸会是`'S'`、`'M'`、`'L'`，所以用 `char` 类型声明变量。如果要在程序中，利用 `Clothes` 类作为设计图，建立衣服实例，要使用 **new** 关键词。例如：

```
new Clothes();
```

在对象术语中，这叫作新建一个对象。如果要有个名牌，专门绑到这个对象上，可以这样声明：

```
Clothes c1;
```

在 Java 的术语中，这叫声明参考名称(Reference Name)、参考变量(Reference Variable)

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

或直接叫参考(Reference)。如果要将 c1 绑到新建的对象上，可以使用 = 指定，以 Java 的术语来说，称为将 c1 名称参考(Refer)至新建对象。例如：

```
Clothes c1 = new Clothes();
```

将程序语法(见图 4.2)直接对应图 4.1，就可以了解类与对象的区别，以及 class、new、=等语法的使用。

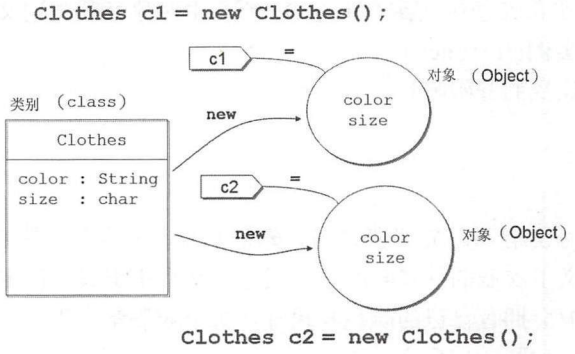


图 4.2 class、new、=等语法对应

提示 »» 对象(Object)与实例(Instance)在 Java 中几乎是等义的名词，本书中就视为相同意义，会交相使用这两个名词。

在 Clothes 类中，定义了 color 与 size 两个变量，以 Java 术语来说，叫作定义两个值域(Field)成员，或叫作定义两个对象数据成员，这表示每个新建的 Clothes 实例，可以拥有个别 color 与 size 值。例如：

ClassObject_Field.java

```
package cc.openhome;

class Clothes {                                    ← ❶ 定义 Clothes 类
    String color;
    char size;
}

public class Field {
    public static void main(String[] args) {
        Clothes sun = new Clothes();
        Clothes spring = new Clothes();            ← ❷ 建立 Clothes 实例

        sun.color = "red";
        sun.size = 'S';
        spring.color = "green";                    ← ❸ 为个别对象的数据成员指定值
        spring.size = 'M';

        System.out.printf("sun (%s, %c)%n", sun.color, sun.size);
        System.out.printf("spring (%s, %c)%n", spring.color, spring.size);   ← ❹ 显示个别对象的数据成员值
    }
}
```

在这个 Field.java 中，定义了两个类，一个是公开(public)的 Field 类，所以文档中的主文档名必须是 Field，另一个是非公开的 Clothes❶。第 2 章中讲过，一个原始码中可以有多个类定义，但只能有一个是公开类，且文档中的主文档名必须与公开类名称相同。

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

提示>>> 只要有一个类定义，编译程序就会产生一个.class 文档。上例中，实际上会产生 Field.class 与 Clothes.class 两个文档。

程序中建立了两个 Clothes 实例，并分别声明了 sun 与 spring 两个名称来参考❷，接着要求 JVM，将 sun 绑定对象上的 color 与 size 分别指定为"red"与'S'，将 spring 的 color 与 size 分别指定为"green"与'M'❸。最后分别显示 sun、spring 的数据成员值❹。

执行结果如下，可以看到 sun 与 spring 各自拥有自己的数据成员：

```
sun (red, S)
spring (green, M)
```

在这个范例中，为个别对象指定数据成员值的程序代码❸，你会发现是类似的。如果想在建立对象时，一并进行某个初始流程，如指定数据成员值，则可以定义构造函数(Constructor)。构造函数是与类名称同名的方法(Method)，直接来看范例比较清楚。

Lab

ClassObject_Field2.java

```
package cc.openhome;

class Clothes2 {
    String color;
    char size;
    Clothes2(String color, char size) {  <-- ❶ 定义构造函数
        this.color = color;  <-- ❷ color 参数的值指定给这个对象的 color 成员
        this.size = size;
    }
}

public class Field2 {
    public static void main(String[] args) {
        Clothes2 sun = new Clothes2("red", 'S');  <-- ❸使用指定构造函数建立对象
        Clothes2 spring = new Clothes2("green", 'M');

        System.out.printf("sun (%s, %c)%n", sun.color, sun.size);
        System.out.printf("spring (%s, %c)%n", spring.color, spring.size);
    }
}
```

在这个例子中，定义新建对象时，必须传入两个自变量给 string 类型的 color 参数(Parameter)与 char 类型的 size 参数❶。而在构造函数中，由于 color 参数与数据成员 color 同名，你不可以直接写 color = color，这是将 color 参数的值指定给 color 参数，而要使用 **this** 表示，将 color 参数的值指定给这个对象(this)的 color 成员。

在实际使用 new 创建对象时，就可以直接传入字符串与字符，分别代表 Clothes 实例的 color 与 size 值，执行结果与上个范例是相同的。

提示>>> 面向对象中有所谓封装(Encapsulation)，不过这里只是定义类，离封装的内涵还有很大的距离，第 5 章会详细介绍如何用 Java 来实现更完整的封装。

4.1.2 使用标准类

在之前介绍的内容中，了解了定义类之后，就可以建立对象进行操作。在第 1 章介绍过，

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Java SE 提供了标准 API，这些 API 就是由许多类所组成的，你可以直接取用这些标准类，省去撰写程序时重新打造基础的需求。下面来举两个基本的标准类：java.util.Scanner 与 java.math.BigDecimal。

提示 >>> java.util 与 java.math 包是归属在 java.base 模块之中，因此不用在模块描述文件中增加任何设定。

1. 使用 java.util.Scanner

目前为止的程序范例的变量值都是写死的，没有办法接受用户的输入。如果要在“命令提示符”模式下取得用户输入，基本上可以使用 **System.in** 对象上的 **read()** 方法，不过这个方法是以 int 类型返回读入的字符编码。想想，如果你输入了一个字符'9'，使用 System.in.read() 的话，就还得自己将字符'9'转换为整数 9，真的是不方便。

可以使用 **java.util.Scanner** 来代劳，下面直接以实际范例来说明。

Lab

ClassObject_Guess.java

```
package cc.openhome;
import java.util.Scanner;  <-- ❶ 告诉编译程序接下来想偷懒

public class Guess {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); <-- ❷ 建立 Scanner 实例
        int number = (int) (Math.random() * 10);
        int guess;

        do {
            System.out.print("猜数字(0 ~ 9):");
            guess = scanner.nextInt(); <-- ❸ 取得下一个整数
        } while(guess != number);

        System.out.println("猜中了...XD");
    }
}
```

如果不想每次都输入 java.util.Scanner，那么一开始就使用 import 告诉编译程序，这样之后就只要输入 Scanner 就可以了❶。在建立 Scanner 实例时，必须传入 java.io.InputStream 的实例，第 10 章介绍输入/输出串流时会提到，System.in 就是一种 InputStream，所以这里可以在创建 Scanner 实例时使用❷。

接下来你想要什么数据，跟 Scanner 对象要就可以了。正如其名，范例中的 Scanner 实例会帮你扫描标准输入，看看用户有无输入字符，怎么扫描你就不用管了，但可以肯定的是有个 Scanner.java 定义了程序代码做这些事，编译为 Scanner.class 成为 Java SE 标准 API 供大家使用。

Scanner 的 **nextInt()** 方法会看看标准输入中，有没有输入下一个字符串(以空格或换行分隔)，有的话会尝试将之剖析为 int 类型❸。Scanner 对每个基本类型都会有个对应的 nextXXX() 方法，如 **nextByte()**、**nextShort()**、**nextLong()**、**nextFloat()**、**nextDouble()**、**nextBoolean()** 等。如果直接取得上一个字符串(以空格或换行分隔)，则使用 **next()**；如果想取得用户输入的整行文字，则使用 **nextLine()**(以换行分隔)。

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示 »» 习惯上，包名称为 java 开头的类，表示标准 API 提供的类。

2. 使用 java.math.BigDecimal

第 2 章介绍基本类型时留了一个问题给你：1.0 - 0.8 的结果是？答案不是 0.2，而是 0.19999999999999996。为什么？这是 Java 的漏洞(Bug)吗？不，不是的。你使用别的程序语言(如 JavaScript、Python 等)也有可能显示这个结果。

简单来说，Java(包括其他程序语言)遵守 IEEE 754 浮点数运算(Floating-Point Arithmetic)规范，使用分数与指数来表示浮点数。例如，0.5 会使用 1/2 来表示，0.75 会使用 1/2 + 1/4 来表示，0.875 会使用 1/2 + 1/4 + 1/8 来表示，而 0.1 会使用 1/16 + 1/32 + 1/256 + 1/512 +1/4096 + 1/8192 + ... 无限循环下去，无法精确表示，因而造成运算上的误差。

再来举个例子，你觉得以下程序片段会显示什么结果？

```
double a = 0.1;
double b = 0.1;
double c = 0.1;
if((a + b + c) == 0.3) {
    System.out.println("等于 0.3");
}
else {
    System.out.println("不等于 0.3");
}
```

由于浮点数误差的关系，结果是显示“不等于 0.3”。类似的例子还很多，结论就是，如果要求精确度，那就要小心使用浮点数，而且别用==直接比较浮点数运算结果。

那要怎么办能得到更好的精确度？可以使用 **java.math.BigDecimal** 类。以刚才的 1.0 - 0.8 为例，如果得到 0.2 的结果，直接使用程序来示范：

ClassObject DecimalDemo.java

```
package cc.openhome;
import java.math.BigDecimal;

public class DecimalDemo {
    public static void main(String[] args) {
        BigDecimal operand1 = new BigDecimal("1.0");
        BigDecimal operand2 = new BigDecimal("0.8");
        BigDecimal result = operand1.subtract(operand2);
        System.out.println(result);
    }
}
```

创建 BigDecimal 的方法之一是使用字符串，BigDecimal 在创建时会剖析传入字符串，以默认精度进行接下来的运算。BigDecimal 提供 **add()**、**subtract()**、**multiply()**、**divide()** 等方法，可以进行加、减、乘、除等运算，这些方法都会返回代表运算结果的 BigDecimal。

上面这个范例可以显示出 0.2 的结果，再来看利用 BigDecimal 比较相等的例子。

Lab

ClassObject DecimalDemo2.java

```
package cc.openhome;

import java.math.BigDecimal;
```

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
public class DecimalDemo2 {
    public static void main(String[] args) {
        BigDecimal op1 = new BigDecimal("0.1");
        BigDecimal op2 = new BigDecimal("0.1");
        BigDecimal op3 = new BigDecimal("0.1");
        BigDecimal result = new BigDecimal("0.3");
        if(op1.add(op2).add(op3).equals(result)) {
            System.out.println("等于 0.3");
        }
        else {
            System.out.println("不等于 0.3");
        }
    }
}
```

由于 BigDecimal 的 add()等方法都会返回代表运算结果的 BigDecimal，所以就直接利用返回的 BigDecimal 再调用 add()方法，最后再调用 **equals()**比较两个 BigDecimal 实质上是否相同，因此有了 a.add(b).add(c).equals(result)的写法。

提示 »» JWorld@TW 的“[FAQ] 为何 1.0 - 0.8 不是 0.2?”讨论中，可以看到更多浮点数误差的例子：
http://www.javaworld.com.tw/jute/post/view?bid=29&id=19197&tpg=1&ppg=1&sty=1&age=0

这里先简介了 BigDecimal 的使用，在第 15 章中还会有更详细的说明。

4.1.3 对象指定与相等性

在上一个范例中，比较两个 BigDecimal 是否相等，是使用 equals()方法而非使用==运算符，为什么？前面提过，Java 并非完全的面向对象程序语言，在 Java 中有两大类型系统，即基本类型与类类型，这很令人困扰，若不讨论底层内存实际运作，初学者就必须区分=与==运算用于基本类型与类类型的不同。

当=用于基本类型时，是将值复制给变量，当==用于基本类型时，是比较两个变量储存的值是否相同，这对初学者来说没有问题。所以下面的程序片段会显示两个 true，因为 a 与 b 储存的值都是 10，而 a 与 c 储存的值也都是 10。

```
jshell> int a = 10;
a ==> 10

jshell> int b = 10;
b ==> 10

jshell> int c = 10;
c ==> 10

jshell> a == b;
$4 ==> true

jshell> a == c;
$5 ==> true
```

如果你在操作对象，=是用在指定参考名称参考某个对象，而==是用在比较两个参考

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

名称是否参考同一对象。对初学者来说，通常看不懂这句话是什么意思。通俗地说，=是用在将某个名牌绑到某个对象，而==是用在比较两个名牌是否绑到同一对象。来看个范例：

```
jshell> BigDecimal a = new BigDecimal("0.1");
a ==> 0.1

jshell> BigDecimal b = new BigDecimal("0.1");
b ==> 0.1

jshell> a == b;
$4 ==> false

jshell> a.equals(b);
$5 ==> true
```

上面的程序片段，建议初学者以绘图方式表示。以第一行为例，看到 new 关键词，就是建立对象，那就画个圆圈表示对象，这个对象内含"0.1"，并建立一个名牌 a 绑到这个新建立的对象，所以第一行与第二行执行后，可用图 4.3 来表示。

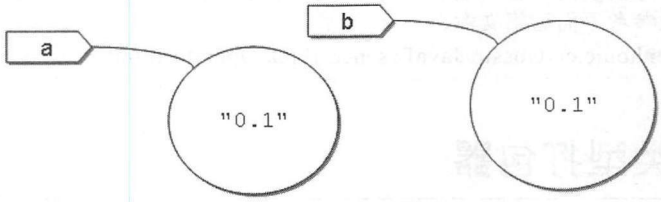


图 4.3　=用于对象指定的示意图

程序中使用 a == b，就是在问，a 名牌绑的对象是否就是 b 名牌绑的对象？答案“不是”，也就是 false 的结果。程序中使用 a.equals(b)，就是在问，a 名牌绑的对象与 b 名牌绑的对象，实际上内含值是否相同？因为 a 与 b 绑的对象，内含值都是"0.1"代表的数值，答案“是”，也就是 true 的结果。

再来看一个例子：

```
jshell> BigDecimal a = new BigDecimal("0.1");
a ==> 0.1

jshell> BigDecimal b = new BigDecimal("0.1");
b ==> 0.1

jshell> BigDecimal c = a;
c ==> 0.1

jshell> a == b;
$4 ==> false

jshell> a == c;
$5 ==> true

jshell> a.equals(b);
$6 ==> true
```

这个程序片段若执行至第三行 c = a，表示将 a 牌子绑的对象也给 c 牌子来绑，可用图 4.4 所示。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

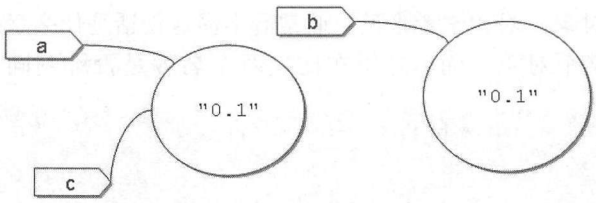


图 4.4 =用于对象指定的示意图

所以问到 a == b，就是在问 a 与 b 是否绑在同一对象？结果就是 false。问到 a == c，就是在问 a 与 c 是否绑在同一对象？结果就是 true。问到 a.equals(b)，就是在问 a 与 b 绑的对象实际上内含值是否相同？结果就是 true。

==用在对象类型，是比较两个名称是否参考同一对象，而!=正好相反，是比较两个名称是否没参考同一对象。实际上，equals()可以自行定义如何比较两对象的内含值，这将在第 6 章中做说明。

提示 »» 其实从内存的实际运作来看，=与==并没有用在基本类型与对象类型的不同，有兴趣的话，可以参考下面这篇文章：
http://openhome.cc/Gossip/JavaEssence/EqualOperator.html

4.2 基本类型打包器

基本类型 long、int、double、float、boolean 等，在 J2SE 5.0 之前必须使用 Long、Integer、Double、Float、Boolean 等类打包为对象，才能当作对象来操作。即使 J2SE 5.0 开始支持了自动装箱(Autoboxing)、拆箱(Unboxing)，仍有必要了解如何打包基本类型，这有助于进一步了解对象与基本类型的差别。

4.2.1 打包基本类型

从第 3 章就一直谈到，Java 中有两个类型系统，即基本类型与类类型。使用基本类型目的在于效率，然而更多时候，会使用类建立实例，因为对象本身可以携带更多信息。如果要让基本类型像对象一样操作，可以使用 **Long、Integer、Double、Float、Boolean、Byte** 等类来打包(Wrap)基本类型。

Long、Integer、Double、Float、Boolean 等类是所谓的打包器(Wrapper)，正如此名称所示，这些类的主要目的就是提供对象实例作为“壳”，将基本类型打包在对象之中，这样就可以操作这个对象，就像是将基本类型当作对象操作。来看个简单的例子：

Wrapper IntegerDemo.java

```
package cc.openhome;

public class IntegerDemo {
    public static void main(String[] args) {
        int data1 = 10;
        int data2 = 20;
        Integer wrapper1 = new Integer(data1); ←— ❶ 打包基本类型
```


严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        Integer wrapper2 = new Integer(data2);
        System.out.println(data1 / 3); ←—— ❷基本类型运算
        System.out.println(wrapper1.doubleValue() / 3); ←—— ❸操作打包器方法
        System.out.println(wrapper1.compareTo(wrapper2));
    }
}
```

基本类型打包器都是归类于 java.lang 包中，如果要使用 Integer 打包 int 类型数据，方法之一是用 new 创建 Integer 实例时，传入 int 类型数据❶。在第 3 章中提到过，如果表达式中都是 int，就只会在 int 空间中做运算，结果会是 int 整数，因此 data1 / 3 就会显示 3 的结果❷。你可以操作 Integer 的 **doubleValue()**将打包值以 double 类型返回，这样就会在 double 空间中做相除，结果就会显示 3.33333333333……❸。

Integer 提供 **compareTo()**方法，可与另一个 Integer 对象进行比较，如果打包值相同就返回 0，小于 compareTo()传入对象打包值就返回-1，否则就是 1。==或!=只能比较是否相等或不相等，而 compareTo()方法可以返回更多信息。

4.2.2 自动装箱、拆箱

除了使用 new 创建基本类型打包器之外，从 J2SE 5.0 之后提供了自动装箱(Autoboxing)功能。可以这样打包基本类型：

```
Integer wrapper = 10;
```

编译程序会自动判断是否能进行自动装箱，在上例中你的 wrapper 会参考 Integer 实例；同样的动作可适用于 boolean、byte、short、char、long、float、double 等基本类型，分别会使用对应的 Boolean、Byte、Short、Character、Long、Float 或 Double 打包基本类型。若使用自动装箱功能来改写 IntegerDemo 中的程序代码：

```
Integer data1 = 10;
Integer data2 = 20;
System.out.println(data1.doubleValue() / 3);
System.out.println(data1.compareTo(data2));
```

程序看起来简洁许多，data1 与 data2 在运行时会参考 Integer 实例，可以直接进行对象操作。自动装箱运用的方法还可以如下：

```
int i = 10;
Integer wrapper = i;
```

也可以使用更一般化的 Number 类来自动装箱。例如：

```
Number number = 3.14f;
```

3.14f 会先被自动装箱为 Float，然后指定给 number。

J2SE 5.0 后可以自动装箱，也可以自动拆箱(Auto Unboxing)，也就是自动取出打包器中的基本形态信息。例如：

```
Integer wrapper = 10;       // 自动装箱
int foo = wrapper;          // 自动拆箱
```

wrapper 会参考至 Integer，若被指定给 int 型的变量 foo，则会自动取得打包的 int 类型再指定给 foo。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

在运算时，也可以进行自动装箱与拆箱。例如：

```
jshell> Integer number = 10;
number ==> 10

jshell> System.out.println(number + 10);
20

jshell> System.out.println(number++);
10
```

上例中会显示 20 与 10，编译程序自动装箱与自动拆箱，也就是 10 会先装箱，然后在 `i + 10` 时会先对 `i` 拆箱，再进行加法运算；`i++`该行也是先对 `i` 拆箱再进行递增运算。再来看一个例子：

```
jshell> Boolean foo = true;
foo ==> true

jshell> System.out.println(foo && false);
false
```

同样地，`foo` 会参考至 `Boolean` 实例，在进行`&&`运算时，会先将 `foo` 拆箱，再与 `false` 进行`&&`运算，结果会显示 `false`。

使用 **new** 建立基本类包是不需要也不建议的，从 Java SE 9 开始，基本类包的建构式都被标示为弃用（Deprecated）了，因此试图使用 **new** 建构基本类包的话会发生编译警告：

```
Note: IntegerDemo.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details
```

在使用 **javac** 时加上**-Xlint:deprecation**，可以看到更清楚的警告细节，例如编译前面的 IntegerDemo.java 的话，会有以下信息：

```
IntegerDemo.java:8: warning: [deprecation] Integer(int) in Integer has been deprecated
        Integer wrapper1 = new Integer(data1);
                           ^
IntegerDemo.java:9: warning: [deprecation] Integer(int) in Integer has been deprecated
        Integer wrapper2 = new Integer(data2);
                           ^
2 warnings
```

4.2.3 自动装箱、拆箱的内幕

自动装箱与拆箱的功能事实上是编译程序蜜糖(Compiler Sugar)，也就是编译程序让你撰写程序时吃点甜头，编译时期根据所撰写的语法，决定是否进行装箱或拆箱动作。例如：

```
Integer number = 100;
```

在 Oracle/Sun 的 JDK 上，编译程序会自动将程序代码展开为：

```
Integer localInteger = Integer.valueOf(100);
```

提示 »» Java 的位码格式也是公开标准，有位码文档，就可以尝试使用反编译程序转译为 Java 语法。本书使用的反编译程序有 JD(http://jd.benow.ca/)与 JAD(http://varaneckas.com/jad/)，后者虽然已不再维护了，不过反编译后的程序代码可看到较多语法蜜糖的实际细节。

使用 `Integer.valueOf()` 也是为基本类型建立打包器的方式之一。了解编译程序如何装箱与拆箱是必要的，例如下面的程序是可以通过编译的：

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
Integer i = null;
int j = i;
```

但是在执行时期会有错误，因为编译程序会将之展开为：

```
Object localObject = null;
int i = localObject.intValue();
```

在 Java 程序代码中，**null** 代表一个特殊对象，任何类声明的参考名称都可以参考至 null，表示该名称没有参考至任何对象实体，这相当于有个名牌没有任何人佩戴。在上例中，由于 i 并没有参考至任何对象，所以就不可能操作 intValue() 方法，就相当于有个名牌没有人佩戴，你却要求戴名牌的人举手，这是一种错误，在 Java 中会出现 **NullPointerException** 的错误信息。

编译程序蜜糖通常提供了方便性，但也因此隐藏了一些细节，所以别只顾着吃糖而忽略了该知道的概念。来看看，下面这段程序，结果会如何？

```
Integer i1 = 100;
Integer i2 = 100;
if (i1 == i2) {
    System.out.println("i1 == i2");
}
else {
    System.out.println("i1 != i2");
}
```

如果只看 Integer i1 = 100，就好像在看 int i1 = 100，直接使用==进行比较，有的人会理所当然回答显示 i1 == i2，那么下面这个呢？

```
Integer i1 = 200;
Integer i2 = 200;
if (i1 == i2) {
    System.out.println("i1 == i2");
}
else {
    System.out.println("i1 != i2");
}
```

注意，程序代码只不过将 100 改为 200，但执行结果会显示 i1 != i2，这是为何？前面提过，自动装箱是编译程序蜜糖。以上例来说，实际上会使用 **Integer.valueOf()** 来建立 Integer 实例，所以你要知道 Integer.valueOf() 到底如何建立 Integer 实例。查看 JDK 文件夹 lib/src.zip 中的 java.base/java/lang 文件夹中的 Integer.java，你会看到 valueOf() 的操作内容：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

这段程序代码简单来说，就是如果传入的 int 在 IntegerCache.low 与 IntegerCache.high 之间，那就尝试看看前面缓存(Cache)中有没有打包过相同的值，如果有就直接返回，否则就使用 new 创建新的 Integer 实例。IntegerCache.low 默认值是-128，IntegerCache.high 默认值是 127。

提示 »» IntegerCache 是 Integer 类别内部实作中的一个类，缓存会在首次使用到 IntegerCache 类时建立。

所以如果是这个程序代码：

```
Integer i1 = 100;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
Integer i2 = 100;
```

第一行程序代码由于100在-128~127之间，会从缓存中传回Integer实例，第二行程序代码执行时，要打包的同样是100，也是从缓存中返回同一Integer实例，所以i1与i2会参考到同一个Integer实例，使用==比较就会是true。

如果是这个程序代码：

```
Integer i1 = 200;
Integer i2 = 200;
```

第一行程序代码由于200不在-128~127之间，所以直接建立Integer实例，第二行程序代码执行时，也是直接建立新的Integer实例，所以i1与i2不会参考到同一个Integer实例，使用==比较就会是false。

IntegerCache.low默认值是-128，执行时期无法更改，IntegerCache.high默认值是127，可以在启动JVM时，使用系统属性**java.lang.Integer.IntegerCache.high**来指定。例如：

```
> java -Djava.lang.Integer.IntegerCache.high=300 cc.openhome.Demo
```

像上面那样指定之后，Integer.valueOf()就会针对-128~300范围中建立的打包器进行快速存取，而针对前面i1与i2打包200时，使用==比较的结果，就又显示i1==i2了。

在IDE中，也可以指定JVM启动时可用的一些自变量。例如在NetBeans中，可以这样操作进行设定：

(1) 在项目上右击，在弹出的快捷菜单中选择“属性”命令，打开“项目属性”对话框，在“类别”列表中选择“运行”节点。

(2) 单击“配置”列表框后面的“新建”按钮，打开“创建新的配置”对话框，在“配置名称”文本框中输入配置文件名称，然后单击“确定”按钮。

(3) 在“主类”文本框中输入与程序进入点的类完全吻合名称。

(4) 在“VM选项”文本框中输入-Djava.lang.Integer.IntegerCache.high=300，单击“确定”按钮完成设定。

这样设定之后，每次在菜单中选择“运行”|“运行主项目”命令时，就会套用“VM选项”中的设定。

所以结论还是4.1节讲过的，别使用==或!=来比较两个对象实质内容值是否相同(因为==与!=是比较对象参考)，而要使用**equals()**。例如以下的程序代码：

```
Integer i1 = 200;
Integer i2 = 200;
if (i1.equals(i2)) {
    System.out.println("i1 == i2");
}
else {
    System.out.println("i1 != i2");
}
```

实际上，无论i1与i2打包的值位于哪个范围，只要i1与i2打包的值相同，equals()比较的结果就会是true。

4.3 数组对象

不管在其他语言中是什么，数组在Java中就是对象，所以前面介绍过的对象基本性质，

在操作数组时也都要注意，如参考名称声明、=指定的作用、==与!=的比较等。掌握这些对象本质，才是灵活操作对象的不二法则。

4.3.1 数组基础

用程序记录 Java 小考成绩，若有 10 名学生，只使用变量的话，必须有 10 个变量储存学生成绩：

```
int score1 = 88;
int score2 = 81;
int score3 = 74;
...
int score10 = 93;
```

实际上不可能这么做，数组基本上是用来收集数据，是具有索引(Index)的数据结构，在 Java 中要声明数组并初始值，可以如下：

```
int[] scores = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
```

这个程序片段建立了一个数组，因为使用 `int[]`声明，所以会在内存中分配长度为 10 的 `int` 连续空间，各个空间储存了 88、81、74、68、78、76、77、85、95、93。各个空间都给予索引编号，索引由 0 开始，由于长度是 10，所以最后一个索引为 9，如果存取超出索引范围，就会抛出 **`ArrayIndexOutOfBoundsException`** 错误。

提示 »» 声明数组时，就 Java 开发人员的撰写习惯来说，建议将`[]`放在类型关键词之后。`[]`也可以放在声明的名称之后，这是为了让 C/C++开发人员看来比较友好，目前来说已不建议：

```
int scores[] = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
```

有关 Java 程序写作的一些惯例，可以参考 Google Java Style：
http:// google.github.io/styleguide/javaguide.html

如果想要依次取出数组中每个值，方法之一是使用 `for` 循环。

Array_Score.java

```
package cc.openhome;

public class Score {
    public static void main(String[] args) {
        int[] scores = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
        for(int i = 0; i < scores.length; i++) {
            System.out.printf("学生分数：%d %n", scores[i]);
        }
    }
}
```

在声明的参考名称旁加上`[]`并指定索引，就可以取得对应值，上例从 `i` 为 0~9，逐一取得值并显示出来。执行结果如下：

```
学生分数：88
学生分数：81
学生分数：74
学生分数：68
学生分数：78
学生分数：76
```

```
学生分数：77
学生分数：85
学生分数：95
学生分数：93
```

在 Java 中数组是对象，而不是单纯的数据集合，数组的 **length** 属性可以取得数组长度，也就是数组的元素个数。

在上面这个范例中，实际上并没有真正需要索引值，索引只是从头递增到尾。如果需求是循序地从头至尾取出数组值，从JDK5之后，有了更方便的增强式 **for** 循环(Enhanced for Loop)语法：

```
for(int score : scores) {
    System.out.printf("学生分数：%d %n", score);
}
```

这个程序片段会取得 scores 数组第一个元素，指定给 score 变量后执行循环体，接着取得 scores 中第二个元素，指定给 score 变量后执行循环体。依此类推，直到 scores 数组中所有元素都访问完为止。将这段 for 循环片段取代 Scores 类中的 for 循环，执行结果相同。实际上，增强式 for 循环是编译程序蜜糖，将位码反编译后可以看到，还是使用索引方式来访问数组：

```
int ai[] = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
int ai1[] = ai;
int i = ai1.length;
for(int j = 0; j < i; j++) {
    int k = ai1[j];
    ...
}
```

如果要设定值给数组中某个元素，也是通过索引。例如：

```
scores[3] = 86;
System.out.println(scores[3]);
```

上面这个程序片段将数组中第 4 个元素(因为索引从 0 开始，索引 3 就是第 4 个元素)指定为 86，所以会显示 86 的结果。

一维数组使用一个索引存取数组元素，二维数组使用两个索引存取数组元素。例如，声明数组来储存 XY 坐标位置要放的值：

Lab

Array_XY.java

```
package cc.openhome;

public class XY {
    public static void main(String[] args) {
        int[][] cords = {
            {1, 2, 3},
            {4, 5, 6}
        };
        for(int x = 0; x < cords.length; x++) {
            for(int y = 0; y < cords[x].length; y++) {
                System.out.printf("%2d", cords[x][y]);
            }
            System.out.println();
        }
    }
}
```

❶ 声明二维数组并赋初始值

❷ 得知有几列

❸ 取得每列的长度

❹ 指定列、行索引取得数组元素

要声明二维数组，就是在类型关键词旁加上[][]❶。初学者暂时将二维数组看作矩阵会比较容易理解，由于有两个维度，必须先通过 cords.length 得知有几列(Row)❷。对于每一列，再利用 cords[x].length 得知每列有几个元素❸。由于在这个范例中，是用二维数组来记录 x、y 坐标的储存值，x、y 就相当于列、行(Column)索引，因此可使用 cords[x][y]来取得 x、y 坐标的储存值。执行结果如下：

```
1 2 3
4 5 6
```

其实这个范例也是循序地走访二维数组，并没有真正要用索引做什么事。用增强式 for 循环来改写会比较简洁：

```
for(int[] row : cords) {
    for(int value : row) {
        System.out.printf("%2d", value);
    }
    System.out.println();
}
```

将这个程序片段取代 XY 类中的 for 循环，执行结果相同，但第一个 for 中的 int[] row : cords 是怎么回事？如果你想知道答案，就得认真了解数组是对象这件事，而不仅仅将它当作连续内存空间。

提示 »» 如果要声明三维数组，就是在类型关键词旁使用[][][]，四维就是[][][][]，依此类推。不过不建议以三维数组以上方式记录数据，因为不容易撰写、阅读与理解，自定义类来解决这类需求会是较好的方式。

4.3.2 操作数组对象

前面都是事先知道元素值建立数组的例子，如果事先不知道元素值，只知道元素个数，那可以使用 new 关键词指定长度来建立数组。例如，预先建立长度为 10 的数组：

```
int[] scores = new int[10];
```

在 Java 中只要看到 new，一定就是建立对象，这个语法代表了数组就是对象。使用 new 建立数组后，每个索引元素会有默认值，如表 4.1 所示。

表 4.1 数组元素初始值

数据类型	初始值
Byte	0
Short	0
Int	0
Long	0L
Float	0.0F
Double	0.0D
Char	\u0000
Boolean	false
类	null

如果默认初始值不符合你的需求，则可以使用 **java.util.Arrays** 的 **fill()** 方法来设定新建数组的元素值。例如，将每个学生的成绩默认为 60 分起：

Array_Score2.java

```
package cc.openhome;
import java.util.Arrays;

public class Score2 {
    public static void main(String[] args) {
        int[] scores = new int[10];
        for(int score : scores) {
            System.out.printf("%2d", score);
        }
        System.out.println();
        Arrays.fill(scores, 60);
        for(int score : scores) {
            System.out.printf("%3d", score);
        }
    }
}
```

执行结果如下：

```
 0 0 0 0 0 0 0 0 0 0
 60 60 60 60 60 60 60 60 60 60
```

如果想在 new 数组中一并指定初始值，可以这样撰写，注意不必指定数组长度：

```
int[] scores = new int[] {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
```

数组既然是对象，在本章一开始，对象是根据类而建立的实例，代表建立数组对象的类定义在哪儿？答案是由 JVM 动态产生。在某种程度上，可以将 int[] 这样的写法，看作类名称，这么看待之后，根据 int[] 而声明的变量就是参考名称了。来看看以下这个片段会显示什么？

```
int[] scores1 = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
int[] scores2 = scores1;
scores2[0] = 99;
System.out.println(scores1[0]);
```

因为数组是对象，而 scores1 与 scores2 是参考名称，若将 scores1 指定给 scores2，意思就是将 scores1 参考的对象也给 scores2 参考，第二行被执行后，可以用图 4.5 来表示。

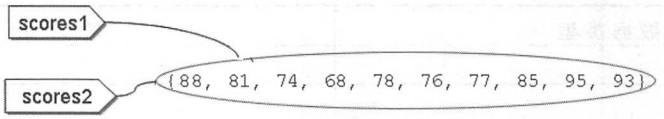


图 4.5　一维数组对象与参考名称

scores2[0] = 99 的意思是，将 scores2 参考的数组对象索引 0 指定为 99，而显示时使用 scores1[0] 的意思是，取得 scores1 参考的数组对象索引 0 的值，结果就是 99。

了解数组是对象，以及 int[] 之类声明的变量就是参考名称之后，来进一步看二维数组。如果想用 new 建立二维数组，则可以如下：

```
int[][] cords = new int[2][3];
```

如果按照一些书籍常用的说法，这是建立了 2×3 的数组，每个索引的默认值如表 4.1 所示，但是这只是简化的说法。这个语法实际上建立了一个 int[][]类型的对象，里面有 2 个 int[]类型的索引，分别是参考长度为 3 的一维数组对象，初始值都是 0，用图 4.6 来表示会更清楚。

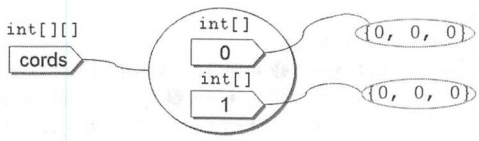


图 4.6　二维数组对象与参考名称

如果将 int[][] cords 看成是 **int[]**[] cords，int[]就相当于一个类 x，实际上你是在声明 x 的一维数组，即 x[]。也就是说，实际上，Java 中的多维数组基本上都是由一维数组组成。

使用 cords.length 取得长度，取得的 cords 参考的对象会有几个索引呢？答案是 2 个。如果问的是 cords[0].length 的值呢？这是在问 cords 的参考对象索引 0 的参考对象(图 4.6 右上的对象)长度是什么？答案就是 3。同理，如果问 cords[1].length 值是什么？这是在问 cords 参考的对象索引 1 的参考对象(图 4.6 右下的对象)长度是什么？答案也是 3。回顾一下前面的 xy 类范例，你应该可以知道为何要这样访问二维数组了：

```
for(int x = 0; x < cords.length; x++) {
    for(int y = 0; y < cords[x].length; y++) {
        System.out.printf("%2d", cords[x][y]);
    }
    System.out.println();
}
```

那么这段增强式 for 语法是怎么回事呢？

```
for(int[] row : cords) {
    for(int value : row) {
        System.out.printf("%2d", value);
    }
    System.out.println();
}
```

根据图 4.6，可以知道，实际上 row 参考到的对象就是一维数组对象，外层 for 循环就是循序取得 cords 参考对象的每个索引，将参考到的对象指定给 int[]类型的 row 名称。

如果使用 new 配置二维数组后想要一并指定初始值，则可以这样撰写：

```
int[][] cords = new int[][] {
    {1, 2, 3},
    {4, 5, 6}
};
```

再试着用图 4.7 表示这段程序代码执行后的结果：

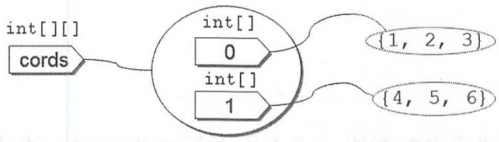


图 4.7　声明二维数组对象与初始值

没有人规定二维数组一定得是矩阵，也可以建立不规则数组。例如：

Lab

Array IrregularArray.java

```
package cc.openhome;

public class IrregularArray {
    public static void main(String[] args) {
        int[][] arr = new int[2][]; ←❶ 声明 arr 参考的对象会有两个索引
        arr[0] = new int[] {1, 2, 3, 4, 5}; ←❷ arr[0]是长度为 5 的一维数组
        arr[1] = new int[] {1, 2, 3};←❸ arr[1]是长度为 3 的一维数组
        for(int[] row : arr) {
            for(int value : row) {
                System.out.printf("%2d", value);
            }
            System.out.println();
        }
    }
}
```

范例中 `new int[2][]`仅提供第一个`[]`数值，这表示 `arr` 参考的对象会有两个索引，但暂时参考至 `null`❶，如图 4.8 所示。

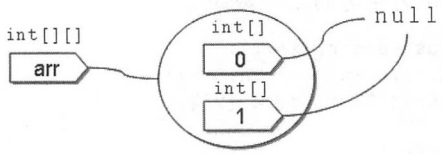


图 4.8　不规则数组示意图之一

接着分别让 `arr[0]`参考至长度为 5 而元素值为 1、2、3、4、5 的数组，以及 `arr[0]`参考至长度为 3 而元素值为 1、2、3 的数组，如图 4.9 所示。

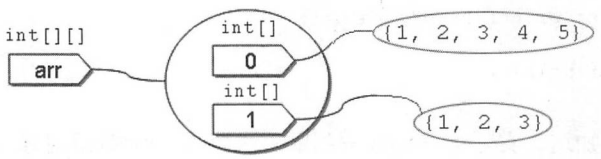


图 4.9　不规则数组示意图之二

这个范例的执行结果如下：

```
 1 2 3 4 5
 1 2 3
```

所以，这么建立数组也是合法的：

```
int[][] arr = {
    {1, 2, 3, 4, 5},
    {1, 2, 3}
};
```

以上都是示范基本类型建立的数组，接下来介绍类类型建立的数组。首先看看如何用 `new` 关键词建立 `Integer` 数组：

```
Integer[] scores = new Integer[3];
```

看来没什么，只不过类型关键词从 `int`、`double` 等换为类名称罢了。那么请问，上面这个

程序片段建立了几个 Integer 对象呢？注意，不是 3 个，是 0 个。回头看一下表 4.1，如果是类类型，这个片段的写法建立的数组，每个索引都是参考至 null，可以用图 4.10 来表示。

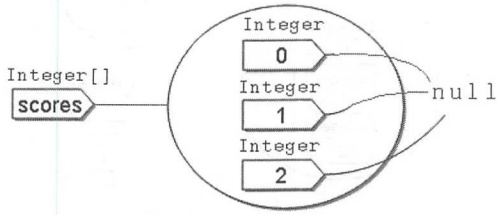


图 4.10　类类型建立的一维数组

每个索引其实都是 Integer 类型，可以参考至 Integer 实例。例如：

Array IntegerArray.java

```
package cc.openhome;
public class IntegerArray {
    public static void main(String[] args) {
        Integer[] scores = new Integer[3];
        for(Integer score : scores) {
            System.out.println(score);
        }
        scores[0] = 99;
        scores[1] = 87;
        scores[2] = 66;
        for(Integer score : scores) {
            System.out.println(score);
        }
    }
}
```

执行结果如下所示：

```
null
null
null
99
87
66
```

将这个片段取代范例中指定索引的部分，执行结果是不变的。如果事先知道 Integer 数组每个元素要放什么，可以如下：

```
Integer[] scores = {99, 87, 66};
```

那么再来问最后一个问题，以下 Integer 二维数组，建立了几个 Integer 实例？

```
Integer[][] cords = new Integer[3][2];
```

应该不会回答 6 个吧！？对初学者来说，建议试着画图来表示，如图 4.11 所示。

new Integer[3][2]代表着一个 Integer[][]类型的对象，其中有 3 个 Integer[]类型的索引，分别参考至长度为 2 的 Integer 一维数组对象，而每个 Integer 一维数组的索引都参考至 null，所以答案还是 0 个 Integer 实例。

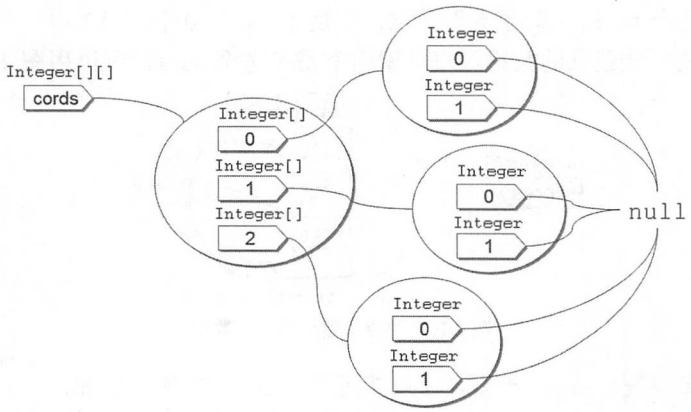


图 4.11　类类型建立的二维数组

4.3.3　数组复制

了解数组是对象，你就应该知道，以下这个并非数组复制：

```
int[] scores1 = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
int[] scores2 = scores1;
```

正如图 4.5 所示，这个程序片段只不过是将 scores1 参考的数组对象，也给 scores2 参考。如果要做数组复制，基本做法是另行建立新数组。例如：

```
int[] scores1 = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
int[] scores2 = new int[scores1.length];
for(int i = 0; i < scores1.length; i++) {
    scores2[i] = scores1[i];
}
```

在这个程序片段中，建立一个长度与 scores1 相同的新数组，再逐一访问 scores1 每个索引元素，并指定给 scores2 对应的索引位置。事实上，不用自行使用循环做值的复制，而可以使用 **System.arraycopy()** 方法，这个方法会使用原生方式复制每个索引元素，比自行使用循环来得快：

```
int[] scores1 = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
int[] scores2 = new int[scores1.length];
System.arraycopy(scores1, 0, scores2, 0, scores1.length);
```

System.arraycopy() 的 5 个参数分别是来源数组、来源起始索引、目的数组、目的起始索引、复制长度。如果使用 JDK6 以上，还有个更方便的 **Arrays.copyOf()** 方法，你不用另行建立新数组，Arrays.copyOf() 会帮你建立。例如：

Array CopyArray.java

```
package cc.openhome;
import java.util.Arrays;

public class CopyArray {
    public static void main(String[] args) {
        int[] scores1 = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
        int[] scores2 = Arrays.copyOf(scores1, scores1.length);
```

```
        for(int score : scores2) {
            System.out.printf("%3d", score);
        }
        System.out.println();
        scores2[0] = 99;
        // 不影响 score1 参考的数组对象
        for(int score : scores1) {
            System.out.printf("%3d", score);
        }
    }
}
```

执行结果如下所示：

```
 88 81 74 68 78 76 77 85 95 93
 88 81 74 68 78 76 77 85 95 93
```

在 Java 中，数组一旦建立，长度就固定了。如果事先建立的数组长度不够怎么办？那就建立新数组，将原数组内容复制至新数组。例如：

```
int[] scores1 = {88, 81, 74, 68, 78, 76, 77, 85, 95, 93};
int[] scores2 = Arrays.copyOf(scores1, scores1.length * 2);
for(int score : scores2) {
    System.out.printf("%3d", score);
}
```

`Arrays.copyOf()`的第二个参数，实际上就是指定建立的新数组长度。上面这个程序片段建立的新数组是 20，执行结果显示原 `scores1` 复制过去的 88 到 93 的元素，然后显示 10 个默认值 0。

以上都是示范基本类型数组，对于类类型声明的数组则要注意参考的行为。来看范例：

Array_ShallowCopy.java

```
package cc.openhome;

class Clothes {
    String color;
    char size;
    Clothes(String color, char size) {
        this.color = color;
        this.size = size;
    }
}

public class ShallowCopy {
    public static void main(String[] args) {
        Clothes[] c1 = {new Clothes("red", 'L'), new Clothes("blue", 'M')};
        Clothes[] c2 = new Clothes[c1.length];
        for(int i = 0; i < c1.length; i++) {
            c2[i] = c1[i];
        }
        c1[0].color = "yellow";
        System.out.println(c2[0].color);
    }
}
```

❶ 复制元素？

❷ 通过 c1 修改索引 0 对象

❸ 通过 c2 取得索引 0 对象的颜色

这个程序的执行结果会是 yellow，这是怎么回事？原因在于循环执行完毕后❶，可用图 4.12 来表示。

实际上循环中仅将 c1 每个索引处所参考的对象，也给 c2 每个索引来参考，并没有实际复制出 Clothes 对象，这叫作复制参考，或称这个行为是浅层复制(Shallow Copy)。无论 **System.arraycopy()**还是 **Arrays.copyOf()**，用在类类型声明的数组时，都是执行浅层复制。如果真的要连同对象一同复制，用户得自行操作，因为基本上只有自己知道，每个对象复制时有哪些属性必须复制。例如：

Lab

Array_DeepCopy.java

```
package cc.openhome;

class Clothes2 {
    String color;
    char size;
    Clothes2(String color, char size) {
        this.color = color;
        this.size = size;
    }
}

public class DeepCopy {
    public static void main(String[] args) {
        Clothes2[] c1 = {new Clothes2("red", 'L'), new Clothes2("blue", 'M')};
        Clothes2[] c2 = new Clothes2[c1.length];
        for(int i = 0; i < c1.length; i++) {
            Clothes2 c = new Clothes2(c1[i].color, c1[i].size); ←—— 自行复制元素
            c2[i] = c;
        }
        c1[0].color = "yellow";
        System.out.println(c2[0].color);
    }
}
```

这个范例执行所谓深层复制(Deep Copy)行为，也就是实际上 c1 每个索引参考的对象会被复制，分别指定给 c2 每个索引，结果就是显示 red。在循环执行完毕后，可用图 4.13 来表示参考与对象之间的关系。

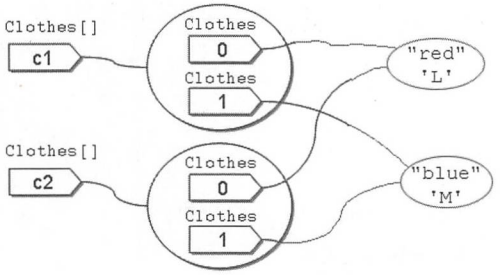


图 4.12　浅层复制

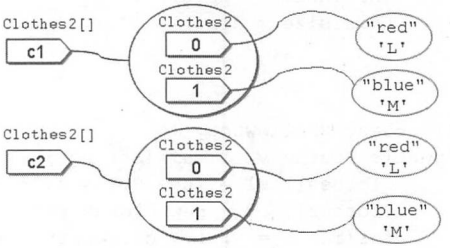


图 4.13　深层复制

4.4　字符串对象

在 Java 中，字符串本质是打包字符数组的对象，是 **java.lang.String** 类的实例。同样地，

前面介绍过的对象操作特性，字符串也都拥有，不过在 Java 中基于效能考虑，给予字符串某些特别且必须注意的性质。

4.4.1 字符串基础

由字符组成的文字符号称为字符串。例如，"Hello"字符串是由'H'、'e'、'l'、'l'、'o'五个字符组成。在某些程序语言中，字符串是以字符数组的方式存在，然而在 Java 中，字符串是 `java.lang.String` 实例，用来打包字符数组。可以用""包括一串字符来建立字符串：

```
jshell> String name = "Justin";
name ==> "Justin"

jshell> name.length();
$2 ==> 6

jshell> name.charAt(0);
$3 ==> 'J'

jshell> name.toUpperCase();
$4 ==> "JUSTIN"
```

由于字符串在 Java 中是对象，所以自然也就拥有一些可操作的方法，如这个程序片段中所示范的，可以使用 **`length()`** 取得字符串长度，使用 **`charAt()`** 指定取得字符串中某个字符，索引从 0 开始，使用 **`toUpperCase()`** 将原本小写的字符串内容转为大写的字符串内容。

如果已经有一个 `char[]` 数组，也可以使用 `new` 来创建 `String` 实例。例如：

```
char[] cs = {'j', 'u', 's', 't', 'i', 'n'};
String name = new String(cs);
```

如果必要，也可以使用 `String` 的 **`toCharArray()`** 方法，以将字符串以 `char[]` 数组返回：

```
char[] cs2 = name.toCharArray();
```

在 Java 中可以使用+运算来连接字符串。例如，在 JDK5 之前并没有 `System.out.printf()` 方法，所以常常看到文字示范在简单的输出时会这么撰写：

```
jshell> String name = "Justin";
name ==> "Justin"

jshell> System.out.println("你的名字：" + name);
你的名字：Justin
```

如果要将字符串转换为整数、浮点数等基本类型，可以使用表 4.2 中类提供的剖析方法。

在表 4.2 中，假设 `number` 参考至 `String` 实例，而该字符串实际上是代表数字，如`"123"`、`"3.14"`这样的数字。如果无法剖析传入的 `String` 实例，则会抛出 **`NumberFormatException`** 的错误。

表 4.2 将字符串剖析为基本类型

方 法	说 明
`Byte.parseByte(number)`	将 number 剖析为 byte 整数
`Short.parseShort(number)`	将 number 剖析为 short 整数

(续表)

方 法	说 明
Integer.parseInt(number)	将 number 剖析为 int 整数
Long.parseLong(number)	将 number 剖析为 long 整数
Float.parseFloat(number)	将 number 剖析为 float 浮点数
Double.parseDouble(number)	将 number 剖析为 double 浮点数

来个综合练习，下面这个范例可以让用户输入整数，输入 0 后会计算所有整数总和并显示：

String_Sum.java

```
package cc.openhome;

import java.util.Scanner;

public class Sum {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        long sum = 0;
        long number = 0;
        do {
            System.out.print("输入数字：");
            number = Long.parseLong(scanner.nextLine());
            sum += number;
        } while(number != 0);
        System.out.println("总和：" + sum);
    }
}
```

执行结果如下：

```
输入数字：10
输入数字：20
输入数字：30
输入数字：0
总和：60
```

为了认识数组与字符串，可以来看看程序进入点 main() 中的 String[] args，在启动 JVM 并指定执行类时，可以一并指定命令行自变量(Command Line Arguments)。例如，在 String 项目文件夹中执行：

```
> java --module-path build\modules -m cc.openhome/cc.openhome.Average 1 2 3 4
```

上面这个指令代表启动 JVM 并执行 cc.openhome.Average 类，而 Average 类会接受 1、2、3、4 这四个自变量，这四个自变量会收集为 String 数组，由 main() 中的 args 参考。

提示 »» 在 NetBeans 中若是采用模块项目，则在项目上右击，在弹出的快捷菜单中选择“构建”命令后，会在项目文件夹中的 build\modules 文件夹中包含编译好的模块，而 dist 文件夹中会包含建构好的 JAR 文件。

实际来看个应用。下面这个范例可让用户命令行自变量提供整数，计算出所有整数平均。

String Average.java

```
package cc.openhome;

public class Average {
    public static void main(String[] args) {
        long sum = 0;
        for(String arg : args) {
            sum += Long.parseLong(arg);
        }
        System.out.println("平均: " + (float) sum / args.length);
    }
}
```

在 NetBeans 中如果要提供命令行自变量，可以这样进行操作。在 IDE 中，也可以指定 JVM 启动时可用的一些自变量。例如在 NetBeans 中，可以进行以下设定。

(1) 在项目上右击，在弹出的快捷菜单中选择“属性”命令，打开“项目属性”对话框，在“类别”列表中选择“运行”节点，如图 4.14 所示。

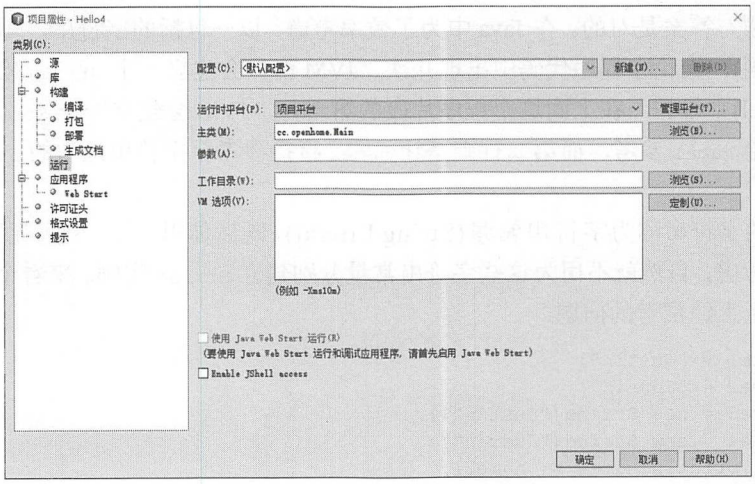


图 4.14 “项目属性”对话框

(2) 单击“配置”列表框后面的“新建”按钮，打开“创建新的配置”对话框，在“配置名称”文本框中输入配置文件名称，然后单击“确定”按钮。

(3) 在“主类”文本框中输入与程序进入点的类完全吻合名称。

(4) 在“参数”文本框中输入命令行自变量，单击“确定”按钮完成设定。

这样设定之后，每次在菜单中选择“运行”|“运行主项目”命令时，就会套用“参数”中的设定。

4.4.2 字符串特性

不同的程序语言会有一些相类似的语法或元素，例如程序语言都会有 `if`、`for`、`while` 之类的语法，也大都有字符、数值、字符串之类的元素。然而各种程序语言解决的问题不同，因此在这些类似语法或元素中，各程序语言会有细微、重要且不容忽视的特性。在学习程序语言时，不得不慎。

以 Java 的字符串来说，就有一些必须注意的特性：

- 字符串常量与字符串池。
- 不可变动(Immutable)字符串。

1. 字符串常量与字符串池

来看个程序片段，你觉得以下会显示 true 还是 false？

```
char[] name = {'J', 'u', 's', 't', 'i', 'n'};
String name1 = new String(name);
String name2 = new String(name);
System.out.println(name1 == name2);
```

希望现在的你有足够的能力与自信回答出 false 的答案，因为 name1 与 name2 分别参考至创建出来的 String 对象。那么下面这个程序代码呢？

```
String name1 = "Justin";
String name2 = "Justin";
System.out.println(name1 == name2);
```

如果你是第一次接触 Java 的话，很意外地，答案会是 true。这代表了 name1 与 name2 是参考到同一对象啰？答案是对的。在 Java 中为了效率考虑，以""包括的字符串，只要内容相同(序列、大小写相同)，无论在程序代码中出现几次，JVM 都只会建立一个 String 实例，并在字符串池(String Pool)中维护。在上面这个程序片段的第一行，JVM 会建立一个 String 实例放在字符串池中，并给 name1 参考，而第二行则是让 name2 直接参考至字符串池中的 String 实例，如图 4.15 所示。

用""写下的字符串称为字符串常量(String Literal)，既然你用"Justin"写死了字符串内容，基于节省内存考虑，自然就不用为这些字符串常量分别建立 String 实例。来看个应用上不会这样撰写，但认证上经常考的问题：

```
String name1 = "Justin";
String name2 = "Justin";
String name3 = new String("Justin");
String name4 = new String("Justin");
System.out.println(name1 == name2);
System.out.println(name1 == name3);
System.out.println(name3 == name4);
```

这个程序片段会分别显示 true、false、false 的结果，因为"Justin"会建立 String 实例并在字符串池中维护，所以 name1 与 name2 参考的是同一个对象，而 new 一定是建立新对象，所以 name3 与 name4 分别参考至新建的 String 实例。以图 4.16 来表示的话就可以知道为何会显示 true、false、false 的结果。

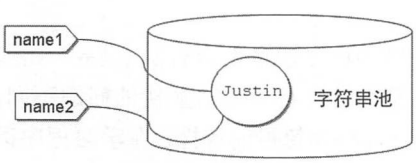


图 4.15　字符串池

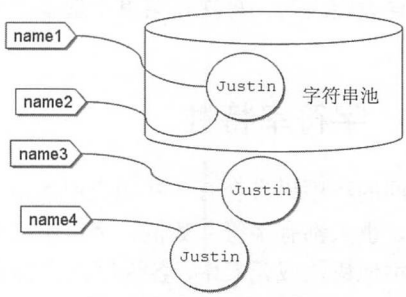


图 4.16　字符串池与新建实例

前面一直强调，如果想比较对象实质内容是否相同，不要使用==，要使用 equals()。同样地，如果想比较字符串实际字符内容是否相同，不要使用==，要使用 **equals()**。以下程序片段执行结果都是显示 true：

```
String name1 = "Justin";
String name2 = "Justin";
String name3 = new String("Justin");
String name4 = new String("Justin");
System.out.println(name1.equals(name2));
System.out.println(name1.equals(name3));
System.out.println(name3.equals(name4));
```

2. 不可变动字符串

在 Java 中，字符串对象一旦建立，就无法更改对象中任何内容，对象上没有任何方法可以更改字符串内容。那么使用+连接字符串是怎么达到的？例如：

```
String name1 = "Java";
String name2 = name1 + "World";
System.out.println(name2);
```

上面这个程序片段会显示 JavaWorld，由于无法更改字符串对象内容，所以绝不是在 name1 参考的字符串对象之后附加 World 内容。可以试着反编译这段程序，结果会发现：

```
String s = "Java";
String s1 = (new StringBuilder()).append(s).append("World").toString();
System.out.println(s1);
```

如果使用+连接字符串，会变成建立 **java.lang.StringBuilder** 对象，使用其 append()方法来进行+左右两边字符串附加，最后再转换为 toString()返回。

简单地说，使用+连接字符串会产生新的 **String** 实例，这并不是告诉你，不要使用+连接字符串，毕竟+连接字符串很方便。这只是在说明，不要将+用在重复性的连接场合，如循环中或递归时使用+连接字符串，这会因为频繁产生新对象，造成效能上的负担。

举个例子来说，如果使用程序显示图 4.17 所示的结果，你会怎么写呢？

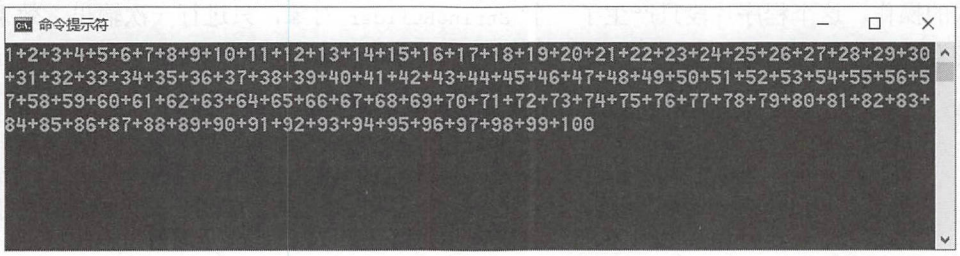


图 4.17　显示 1+2+...+100

这是个很有趣的题目，以下列出几个我看过的写法。首先有人这么写：

```
for(int i = 1; i < 101; i++) {
    System.out.print(i);
    if(i != 100) {
        System.out.print('+');
    }
}
```

这可以达到题目要求，不过有没有改进的空间？其实可以改成这样：

```
for(int i = 1; i < 100; i++) {
    System.out.printf("%d+", i);
}
System.out.println(100);
```

程序变简洁了，而且可以少一个 if 判断，不过就这个小程序而言，少个 if 判断节省不了多少时间。事实上，可以减少输出次数，因为在 for 循环中调用了 99 次 System.out.printf()，相较于内存中的运算，标准输出速度是慢得多了。有人知道可以使用+连接字符串，所以会这么写：

```
String text = "";
for(int i = 1; i < 100; i++) {
    text = text + i + '+';
}
System.out.println(text + 100);
```

在这个程序片段中，for 循环中没有进行输出，这确实提高了效率，不过在 Java 中使用+连接会产生新字符串对象，这个程序片段 for 循环中有频繁产生新对象的问题，正如前面对+连接片段反编译中所看到的，可以改用 StringBuilder 来改善。

String_OneTo100.java

```
package cc.openhome;

public class OneTo100 {
    public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();
        for (int i = 1; i < 100; i++) {
            builder.append(i).append('+');
        }
        System.out.println(builder.append(100).toString());
    }
}
```

StringBuilder 每次 append() 调用过后，都会返回原有的 StringBuilder 对象，方便你进行下一次的操作。这个程序片段只产生了一个 StringBuilder 对象，只进行一次输出，效果上对比最初看到的程序片段好得多。

提示 »» java.lang.StringBuilder 是 JDK5 之后新增的类，在该版本之前，是使用 java.lang.StringBuffer 类，StringBuilder 与 StringBuffer 具有相同操作接口。在单机非多线程(Multithread)情况下，使用 StringBuilder 会有较好的效率，因为 StringBuilder 不处理同步(Synchronized)问题；StringBuffer 则会处理同步问题。在多线程环境下建议改用 StringBuffer，让对象自行管理同步问题。第 11 章还会介绍何为多线程。

再来看下面的例子，请问以下会显示 true 还是 false？

```
String text1 = "Ja" + "va";
String text2 = "Java";
System.out.println(text1 == text2);
```

有的人会这么说：因为用了+连接字符串，产生新字符串，所以 text1 == text2 应该是 false 吧。如果你这么认为，那就上当了。答案是 true。反编译之后就知道为什么了：

```
String s = "Java";
String s1 = "Java";
System.out.println(s == s1);
```

编译程序是这么认为的：既然你写死了"Ja" + "va"，那你要的不就是"Java"吗？根据以上反编译之后的程序代码，显示 true 的结果就不足为奇了。

4.4.3 字符串编码

前面字符串都用英文示范，但日后在 Java 中不会只处理英文，所以要了解 Java 中如何处理中文。在正式介绍 Java 如何处理字符串编码前，容我问一句：你写的.java 原始码文档是什么编码？

这其实是个简单但蛮重要的问题，可事实上有许多人答不出来。如果是简体中文 Windows，建立纯文本文档重新命名为.java，使用 Windows 默认的纯文本编辑器，那么应该是兼容于 MS950 编码（兼容于 Big5）。如果在 NetBeans 中建立.java 原始码，默认应该是 UTF-8 编码。如果是在 Eclipse 中建立.java 原始码，而 Eclipse 运行在简体中文 Windows 中，默认应该是 MS950 编码。

知道这个问题的答案后，再继续想想，之前章节介绍过 Java 支持 Unicode，所以写下一个英文字符或写下一个汉字字符，都是双字节。那么你是否考虑过，明明你的 Windows 纯文本编辑器是 MS950 编码，为什么写下的字符串在 JVM 中会是 Unicode？我们在一个 Main.java 中写以下的程序代码并编译：

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello");
        System.out.println("哈啰");
    }
}
```

如果操作系统默认编码是 MS950，而文本编辑器是使用 MS950 编码，那么这样执行编译：

```
> javac Main.java
```

产生的.class 文档，使用反汇编程序还原的程序代码中，会看到以下内容：

```
public class Main {
    public static void main(String args[]) {
        System.out.println("Hello");
        System.out.println("\u54C8\u56C9");
    }
}
```

还记得表 3.2 吗？字符可以用\uxxxx 来表示，其中 xxxx 是字符的 Unicode 编码。上面反编译的程序中，"\u54C8\u56C9"就是"哈啰"的 Unicode 编码表示。JVM 在加载.class 之后，就是读取 Unicode 编码并产生对应的字符串对象，而不是最初在原始码中写下的"哈啰"。

那么编译程序怎么知道要将汉字字符转为哪个 Unicode 编码？当使用 javac 指令没有指定 **-encoding** 选项时，会使用操作系统默认编码。如果文字编译程序是使用 UTF-8 编码，那么编译时就要指定-encoding 为 UTF-8，这样编译程序才会知道用何种编译读取.java 的内容。例如：

```
> javac -encoding UTF-8 Main.java
```

提示 »» 在 Windows 中不要使用纯文本编辑器转存的 UTF-8 文件来撰写 Java 程序，Windows 的纯文本编辑器会在文档头加上 BOM，虽然允许为 UTF-8 文档标识 BOM，但其实不需要。简单地说，javac 看到 BOM 会无法编译。正如第 2 章提示过的，建议可以使用 NotePad++，它可以帮用户很方便地选择文字编码：
http://notepad-plus-plus.org/

IDE 通常可以使用户自定义原始码编码，如果是 NetBeans，可以在项目上右击，在弹出的快捷菜单中选择“属性”命令，打开“项目属性”对话框，在“类别”列表中选择“源”节点，在右侧“编码”列表框中可以选择原始码编码，如图 4.18 所示。

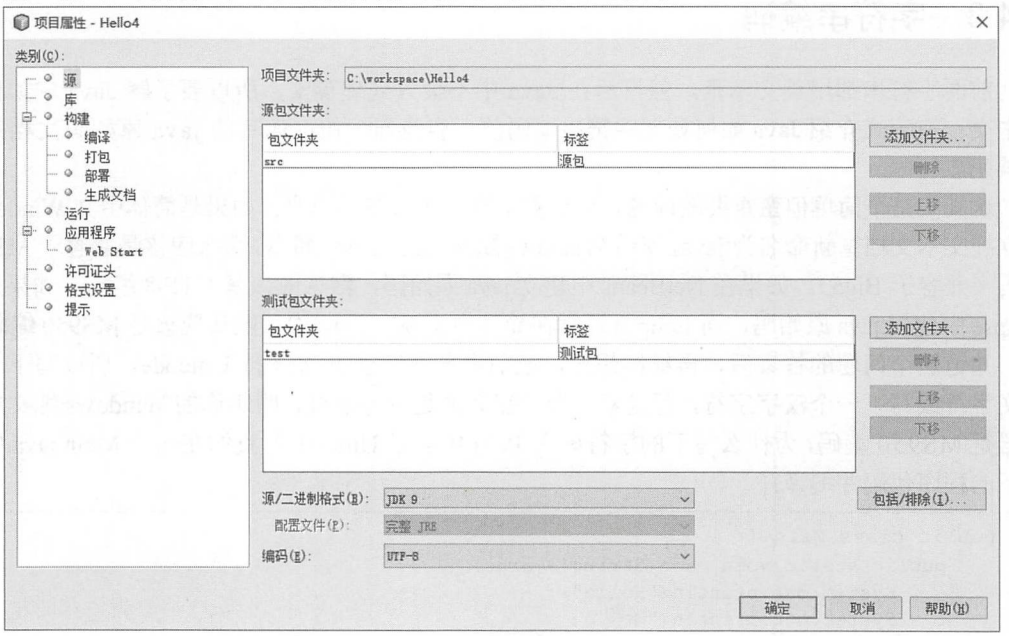


图 4.18　设定原始码编码

4.5　查询 Java API 文件

本章谈到了许多的类，如 `java.util.Scanner`、`java.math.BigDecimal`、各种基本类型打包器、`java.util.Arrays`、`java.lang.String`、`java.lang.StringBuilder` 等，也使用了一些类上定义的方法。那么，如果书上没示范过的方法，你怎么知道如何使用？查询 Java API 文件。

这很重要，所以在这里单独用一个小节来说明。我的写作习惯一向是不爱列出一大堆表格来说明，某某类上有某某方法，因为这应该是你知道如何在网络上查询 API 文件来得知。在书上列出过多表格只是沦为翻译，而且也没意义，因为未来你会遇到难以计数的类与方法，就算是 Java SE 书也不可能列齐所有类与方法说明。

如何查询 Java API 文件？以本书范围来说，将以查询 Java SE API 文件为例。通过下面的链接打开 Oracle JDK 9 Documentation，如图 4.19 所示。

http://docs.oracle.com/javase/9/

可以看到，在这个页面中有许多关于 JDK9 的相关文件，也可以查询到 JDK8 或更早版本的 JDK 文件。想要查看 API 文件的话，可以单击左边的 API Documentation，查看相关文件，如图 4.20 所示。

由于 Java SE 9 支持模块化的关系，在页面中可以看到各个模块名称，在查询 API 文件时，比较方便的查看方式是页框模式，因此可以单击正上方导览中的 FRAMES，如图 4.21 所示。

图 4.19 Oracle JDK 9 Documentation

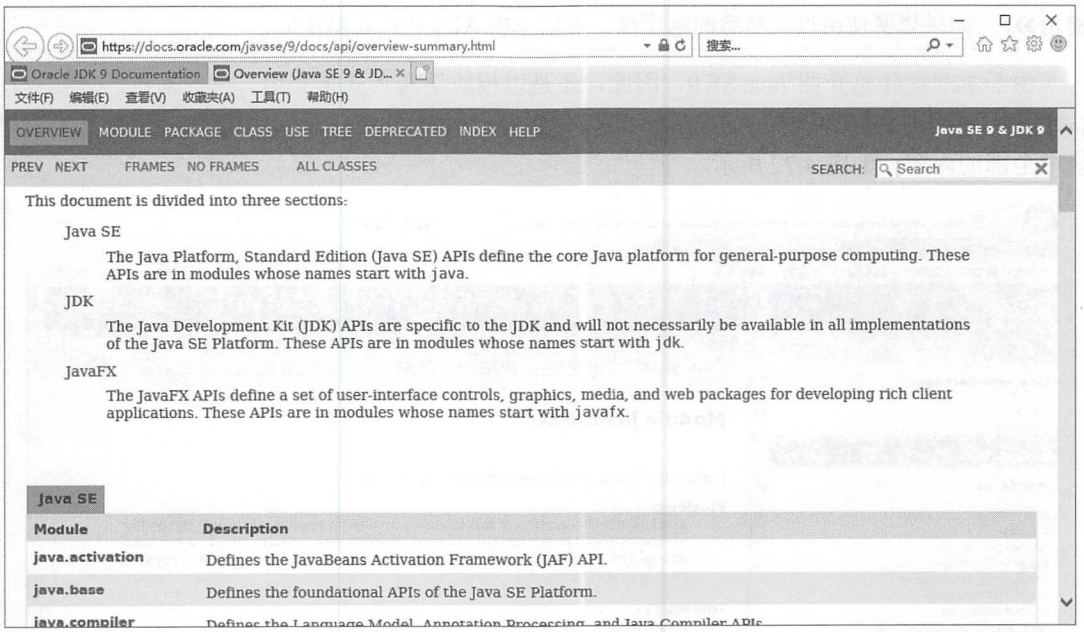


图 4.20 API Documentation

如果过去你曾经有过查询 API 文件的经验，马上会看到不同的地方，左上方默认会显示模层级层而不是包层级。如果将来你已经熟悉哪个包位于哪个模块下面，模块层级形式应该会比较方便。

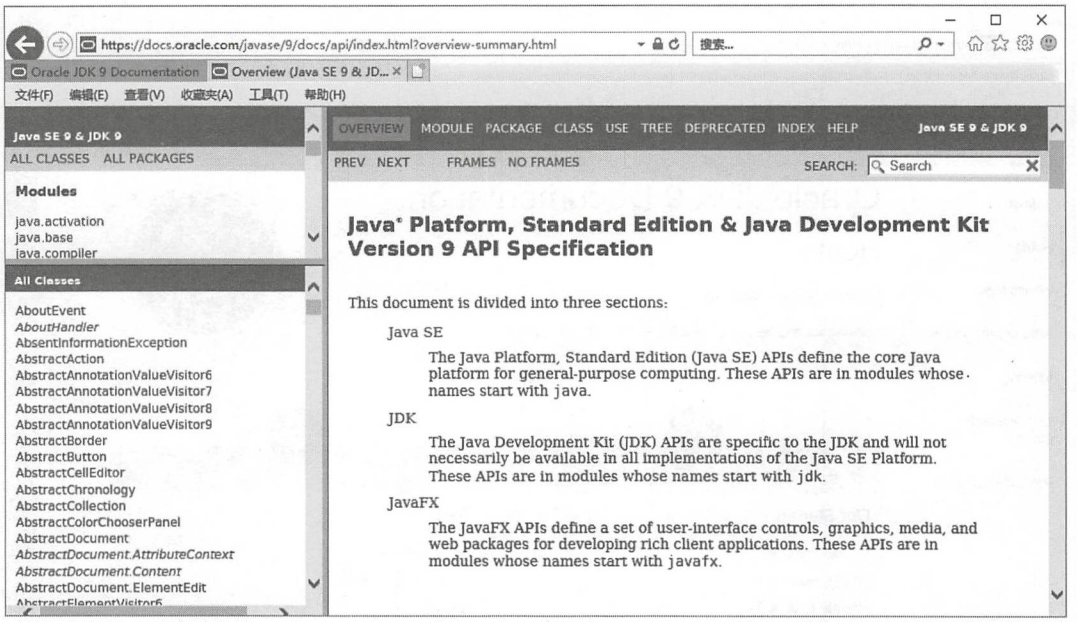


图 4.21　页框显示

提示 »» 如果想要使用过去熟悉的包层级，可以单击 ALL PACKAGES。

由于本书主要是介绍 Java SE 9，因此，还是以模块层级形式来介绍。举例来说，想要查询 `java.base` 模块中的 `java.lang.String`，可以先在左上页框单击 `java.base` 模块，这会显示该模块中全部的内容，如图 4.22 所示。

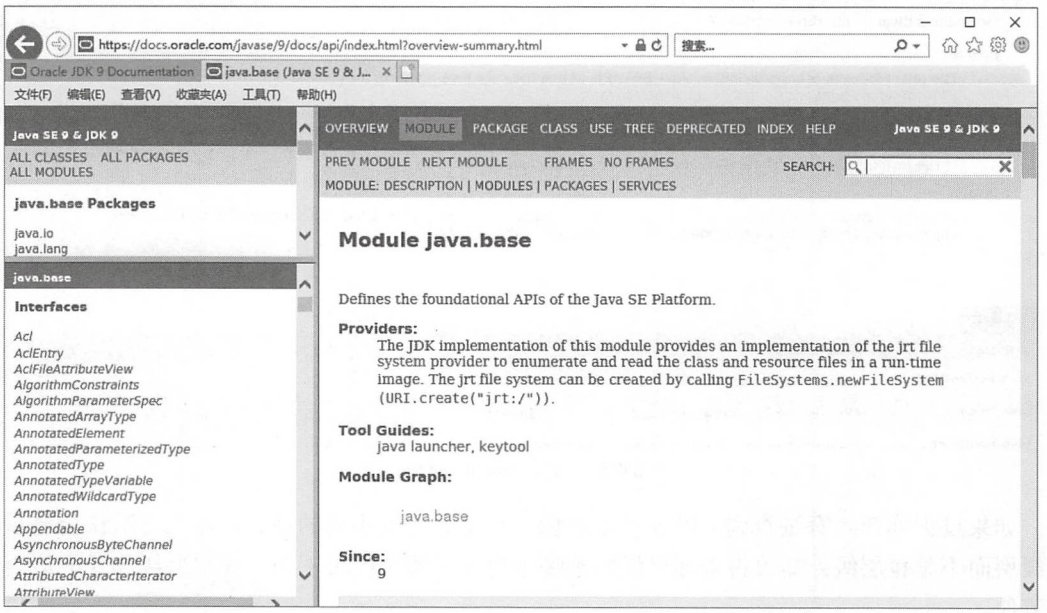


图 4.22　查看 `java.base` 模块

接着再选择 `java.lang`，然后在左下页框选择 `java.lang.String` 类，右边窗格就会显示 `java.lang.String` 的说明，如图 4.23 所示。

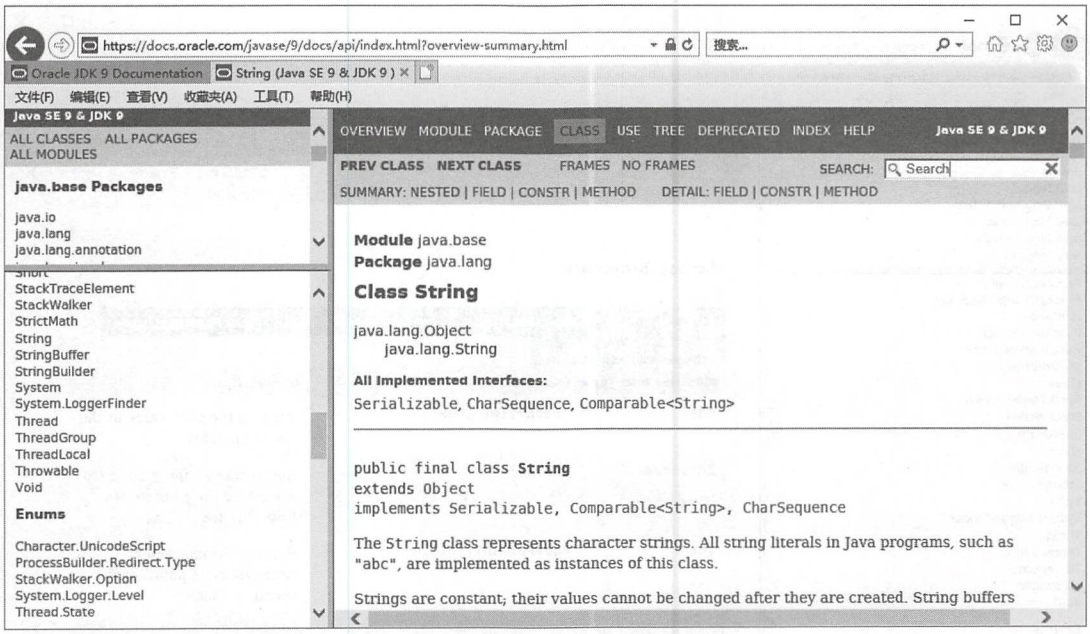


图 4.23 查看 java.lang.String 类

就初学者而言，目前可以先了解构造函数的部分，如图 4.24 所示。

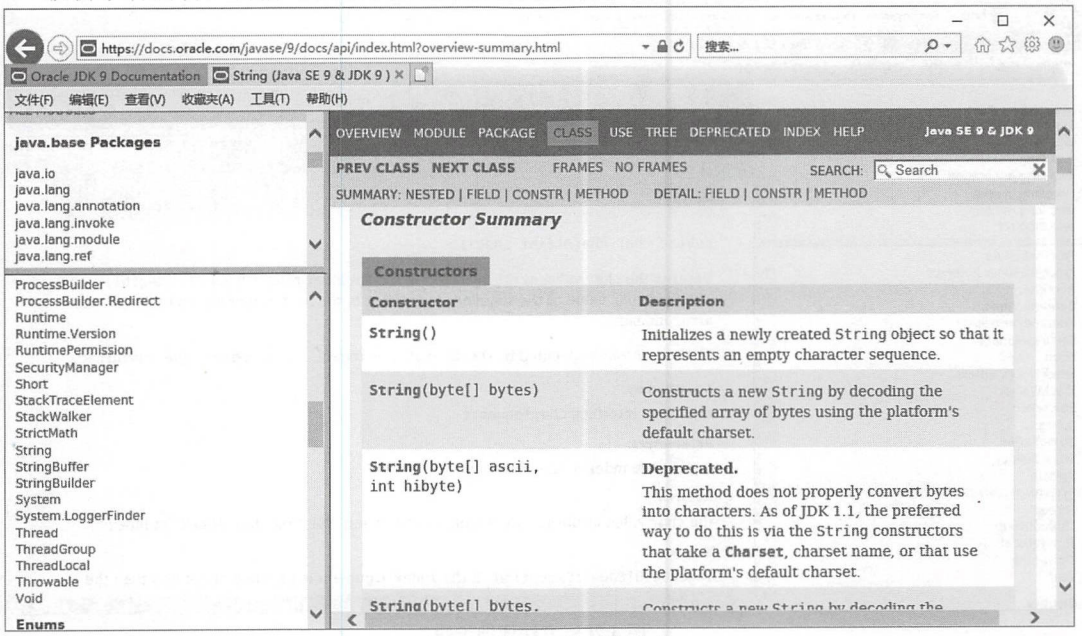


图 4.24 构造函数列表说明

这显示出创建 String 实例时可给予的数据，另一个就是方法说明，如图 4.25 所示。

这显示了可用的方法名称、参数类型与返回类型，单击任一方法链接，还可以看到详细说明，如单击 charAt()方法，如图 4.26 所示。

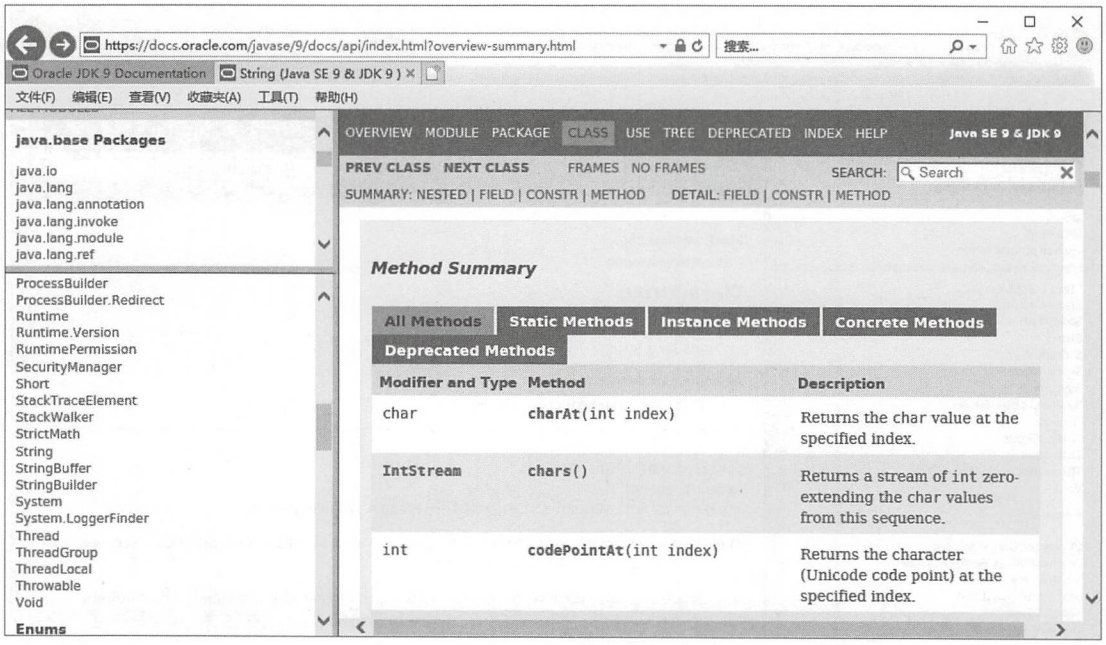


图 4.25　方法列表说明

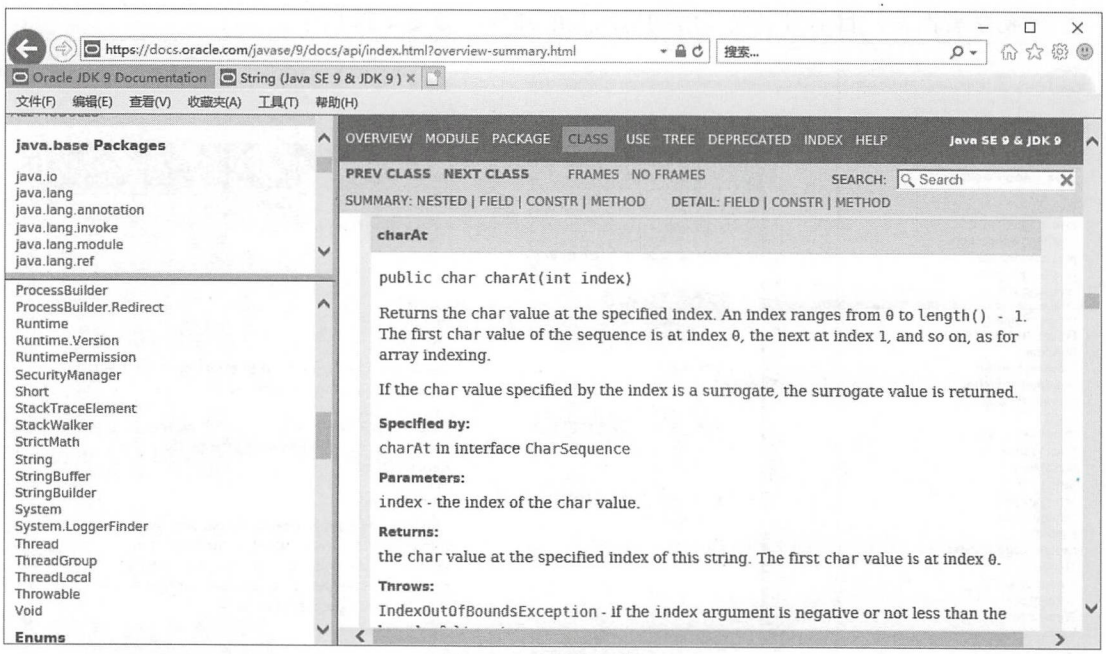


图 4.26　方法详细说明

在 API 文件的右上方，有个 SEARCH 框。Java SE 9 的 API 文件是基于 HTML5 撰写，这个搜索框是从 Java SE 9 开始出现在 API 文件之中，对于搜索 API 来说，提供了很大方便，例如搜索 `BigDecimal` 的相关 API，如图 4.27 所示。

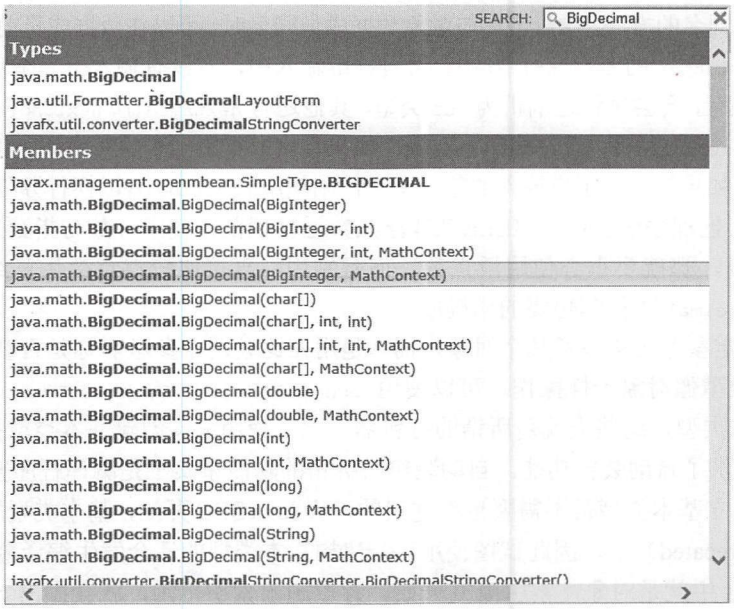


图 4.27　搜索 BigDecimal 的相关 API

现在搜索引擎很方便，只要你知道类的完全吻合名称以及想查询的版本。例如查询 Java SE 9 的 java.lang.String，可在搜索引擎中进行查询，单击之后，就可以看到文件说明了，如图 4.28 所示。

图 4.28　利用搜索引擎查询 Java API 文件

随着你对 Java 语法认识的增多，对 Java API 文件中各个部分的说明也会有更多认识，后面章节若必要，还会再提示 API 文件中记载的相关信息。

4.6　重点复习

要产生对象必须先定义类，类是对象的设计图，对象是类的实例。类定义时使用 class 关键词，建立实例要使用 new 关键词。以类名称声明的变量，称为参考名称、参考变量或直接叫参考。

想在建立对象时，一并进行某个初始流程，如指定数据成员值，则可以定义构造函数。构造

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

函数是与类名称同名的方法。参数名称与对象数据成员同名时，可以在数据成员前使用 this 区别。

java.util.Scanner 的 nextInt()方法会看看标准输入中，有没有输入下一个字符串(以空格或换行分隔)，如果有，会尝试将之剖析为 int 类型，其他还有 nextByte()、nextShort()、nextLong()、nextFloat()、nextDouble()、nextBoolean()等。如果直接取得上一个字符串(以空格或换行分隔)，则使用 next()；如果想取得用户输入的整行文字，则使用 nextLine()(以换行分隔)。

Java(包括其他程序语言)遵守 IEEE 754 浮点数运算规范，使用分数与指数来表示浮点数。如果要求精确度，那就要小心使用浮点数，而且别用==直接比较浮点数运算结果。可以使用 java.math.BigDecimal 类得到想要的精确度。

=是用在指定参考名称参考某个对象，而==是用在比较两个参考名称是否参考同一对象。

要让基本类型像对象一样操作，可以使用 Long、Integer、Double、Float、Boolean、Byte 等类来打包基本类型，这些类就是所谓的打包器。除了使用 new 创建基本类型打包器之外，从 J2SE 5.0 之后提供了自动装箱功能。自动装箱与拆箱的功能事实上是编译程序蜜糖。

使用 new 建立基本类型是不需要也不建议的，从 Java SE 9 开始，基本类型的建构式都被标示为弃用（Deprecated）了，因此试图使用 new 建构基本类型的话会发生编译警告。

数组在 Java 中就是对象，索引由 0 开始，存取超出索引范围，就会抛出 ArrayIndexOutOfBoundsException 错误。从 JDK5 之后，有了更方便的增强式 for 循环语法，可用于循序取得数组元素。使用 new 建立数组后，每个索引元素会有默认值，如表 4.1 所示。在 Java 中，数组一旦建立，长度就固定了。

无论 System.arraycopy()还是 Arrays.copyOf()，用在类类型声明的数组时，都是执行浅层复制。

字符串本质是打包字符数组的对象，是 java.lang.String 类的实例。在启动 JVM 并指定执行类时，可以一并指定命令行自变量，会收集为 String 数组，由 main()中的 args 参考。

以""包括的字符串，只要内容相同(序列、大小写相同)，无论在程序代码中出现几次，JVM 都只会建立一个 String 实例，并在字符串池中维护。如果想比较字符串实际字符内容是否相同，不要使用==，要使用 equals()。

字符串对象一旦建立，就无法更改对象中任何内容，对象上没有任何方法可以更改字符串内容。使用+连接字符串会产生新的 String 实例，不要将+用在重复性的连接场合。

使用 javac 指令没有指定-encoding 选项时，会使用操作系统默认编码。

4.7 课后练习

1. Fibonacci 为 13 世纪欧洲数学家，在他的著作中提过，若一只兔子每月生一只小兔子，一个月后小兔子也开始生产。起初只有一只兔子，一个月后有两只兔子，两个月后有三只兔子，三个月后有五只兔子……，也就是每个月兔子总数会是 1、1、2、3、5、8、13、21、34、55、89……，这就是费氏数列，可用公式定义如下：

$$f_n = f_{n-1} + f_{n-2} \quad \text{if } n > 1$$

$$f_n = n \quad \text{if } n = 0, 1$$

请撰写程序，可让用户输入想计算的费式数个数，由程序全部显示出来。例如：

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
求几个费式数？10
0 1 1 2 3 5 8 13 21 34
```

2. 请撰写一个简单的洗牌程序，可在文本模式下显示洗牌结果。例如：

```
桃 6 砖 9 砖 6 梅 5 梅10 心 5 梅 K 梅 6 心 J 心 1 心 6 梅 3 梅 7
砖 4 砖 1 心 7 砖 2 砖 J 梅 Q 桃 2 心 2 梅 2 心10 桃 7 桃 1 桃 8
心 9 砖 Q 砖 7 心 3 梅 9 梅 1 心 4 桃 Q 桃10 桃 3 砖 K 桃 K 桃 9
砖10 梅 8 砖 3 梅 4 砖 8 砖 5 桃 5 心 8 梅 J 心 Q 桃 J 桃 4 心 K
```

3. 下面是一个数组，请使用程序使其中元素排序为由小到大：

```
int[] number = {70, 80, 31, 37, 10, 1, 48, 60, 33, 80};
```

4. 下面是一个排序后的数组，请撰写程序让用户在数组中寻找指定数字，找到就显示索引值，找不到就显示-1：

```
int[] number = {1, 10, 31, 33, 37, 48, 60, 70, 80};
```


严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 5 对象封装

学习目标

- 了解封装的概念与实现
- 定义类、构造函数与方法
- 使用方法重载与不定长度自变量
- 了解 static 成员

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

5.1 何谓封装

第4章简要介绍了如何定义类，有个概念必须先理清，定义类并不等于做好了面向对象中封装(Encapsulation)的概念。那么到底什么才是封装的含义？此时必须以对象的角度来思考问题。

本节着重在封装的概念，且一并说明如何以Java语法来操作，有一些内容会略微与第4章重复，这是为了介绍上的完整性。在了解封装基本概念之后，下一节会进入Java的语法细节。

5.1.1 封装对象初始流程

假设要写个可以管理储值卡的应用程序，首先得定义储值卡会记录哪些数据，如储值卡号码、余额、红利点数，在Java中可使用 **class** 关键字进行定义：

```
package cc.openhome;
class CashCard {
    String number;
    int balance;
    int bonus;
}
```

假设将这个类定义在 cc.openhome 包，使用 CashCard.java 储存，编译为 CashCard.class，并将这个位码给朋友使用，则你的朋友要建立5张储值卡的数据：

```
CashCard card1 = new CashCard();
card1.number = "A001";
card1.balance = 500;
card1.bonus = 0;

CashCard card2 = new CashCard();
card2.number = "A002";
card2.balance = 300;
card2.bonus = 0;

CashCard card3 = new CashCard();
card3.number = "A003";
card3.balance = 1000;
card3.bonus = 1;  // 单次储值1000元可获得红利一点
...
```

在这里可以看到，如果想存取对象的数据成员，可以通过“.”运算符加上数据成员名称。

可以发现，每次在建立储值卡对象时，都会做相同的初始动作，也就是指定卡号、余额与红利点数，这个流程是重复的，更多的 CashCard 对象建立会带来更多的程序代码重复。在程序中出现重复的流程，往往意味着有改进的空间，在 4.1.1 节中介绍过，Java 中可以定义构造函数(Constructor)来改进这个问题：

Encapsulation1 CashCard.java

```
package cc.openhome;
class CashCard {
    String number;
    int balance;
    int bonus;
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    CashCard(String number, int balance, int bonus) {
        this.number = number;
        this.balance = balance;
        this.bonus = bonus;
    }
}
```

正如 4.1.1 节介绍过的，构造函数是与类名称同名的方法(Method)，不用声明返回类型。在这个例子中，构造函数上的 number、balance 与 bonus 参数，与类的 number、balance、bonus 数据成员同名了。为了区别，在对象数据成员前加上 **this** 关键字，表示将 number、balance 与 bonus 参数的值，指定给这个对象(this)的 number、balance、bonus 数据成员。

在重新编译 CashCard.java 之后，交给你的朋友，同样是建立 5 个 CashCard 对象，现在只要这么写：

```
CashCard card1 = new CashCard("A001", 500, 0);
CashCard card2 = new CashCard("A002", 300, 0);
CashCard card3 = new CashCard("A003", 1000, 1);
...
```

比较看看，他应该会想写这个程序片段，还是刚刚那个程序片段？那么你封装了什么？你用了 Java 的构造函数语法，实现对象初始化流程的封装。封装对象初始化流程有什么好处？拿到 CashCard 类的用户，不用重复撰写对象初始化流程，事实上，他也不用知道对象如何初始化，就算你修改了构造函数的内容，重新编译并给予位码文档之后，CashCard 类的用户也无须修改程序。

实际上，如果你的类用户想建立 5 个 CashCard 对象，并将数据显示出来，可以用数组，而无须个别声明参考名称。例如：

Encapsulation1 CashApp.java

```
package cc.openhome;

public class CardApp {
    public static void main(String[] args) {
        CashCard[] cards = {
            new CashCard("A001", 500, 0),
            new CashCard("A002", 300, 0),
            new CashCard("A003", 1000, 1),
            new CashCard("A004", 2000, 2),
            new CashCard("A005", 3000, 3)
        };

        for(CashCard card : cards) {
            System.out.printf("(%s, %d, %d)%n",card.number, card.balance, card.bonus);
        }
    }
}
```

执行结果如下所示：

```
(A001, 500, 0)
(A002, 300, 0)
(A003, 1000, 1)
(A004, 2000, 2)
(A005, 3000, 3)
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示 »» 接下来说明范例时，都会假设有两个以上的开发者。记住，如果面向对象或设计上的议题对你来说太抽象，请从两人或多人共同开发的角度来想想看，这样的概念与设计对大家合作有没有好处。

5.1.2 封装对象操作流程

假设现在你的朋友使用 CashCard 建立 3 个对象，并要对所有对象进行储值的动作：

```
Scanner scanner = new Scanner(System.in);
CashCard card1 = new CashCard("A001", 500, 0);
int money = scanner.nextInt();
if(money > 0) {
    card1.balance += money;
    if(money >= 1000) {
        card1.bonus++;
    }
}
else {
    System.out.println("储值是负的？你是来乱的吗？");
}

CashCard card2 = new CashCard("A002", 300, 0);
money = scanner.nextInt();
if(money > 0) {
    card2.balance += money;
    if(money >= 1000) {
        card2.bonus++;
    }
}
else {
    System.out.println("储值是负的？你是来乱的吗？");
}

CashCard card3 = new CashCard("A003", 1000, 1);// 还是那些 if...else 的重复流程
...
```

你的朋友做了简单的检查，就是储值不能是负的，而储值大于 1000 的话，就给予红利，很容易就可以发现，那些储值的流程重复了。储值这个动作应该是 CashCard 对象自己处理！在 Java 中，可以定义方法(Method)来解决这个问题：

Encapsulation2 CashCard.java

```
package cc.openhome;
class CashCard {
    String number;
    int balance;
    int bonus;
    CashCard(String number, int balance, int bonus) {
        this.number = number;
        this.balance = balance;
        this.bonus = bonus;
    }

    void store(int money) {  // 储值时调用的方法 ←❶ 不会返回值
```


严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        if(money > 0) {
            this.balance += money;
            if(money >= 1000) {
                this.bonus++;
            }
        }
        else {
            System.out.println("储值是负的？你是来乱的吗？");
        }
    }

    void charge(int money) { // 扣款时调用的方法
        if(money > 0) {
            if(money <= this.balance) {
                this.balance -= money;
            }
            else {
                System.out.println("钱不够啦！");
            }
        }
        else {
            System.out.println("扣负数？这不是叫我储值吗？");
        }
    }

    int exchange(int bonus) {  // 兑换红利点数时调用的方法
        if(bonus > 0) {
            this.bonus -= bonus;
        }
        return this.bonus;
    }
}
```

❷ 封装储值流程

❸ 会返回 int 型态

在 CashCard 类中，除了定义储值用的 store() 方法之外，还要考虑到扣款用的 charge() 方法，以及兑换红利点数的 exchange() 方法。在类中定义方法，如果不用返回值，方法名称前可以声明 **void**❶。

前面看到的储值重复流程，现在都封装到 store() 方法中❷，这么做的好处是使用 CashCard 的用户现在可以这么撰写了：

```
Scanner scanner = new Scanner(System.in);
CashCard card1 = new CashCard("A001", 500, 0);
card1.store(scanner.nextInt());

CashCard card2 = new CashCard("A002", 300, 0);
card2.store(scanner.nextInt());

CashCard card3 = new CashCard("A003", 1000, 1);
card3.store(scanner.nextInt());
```

相比前面的撰写重复流程，CashCard 用户应该会比较想写这个吧！你封装了什么呢？你封装了储值的流程。哪天你也许考虑每加值 1000 元就增加一点红利，而不像现在就算加值 5000 元也只有一点红利，就算改变了 store() 的流程，CashCard 用户也无须修改程序。

同样地，charge() 与 exchange() 方法也分别封装了扣款以及兑换红利点数的流程。为了知道兑换红利点数后，剩余的点数还有多少，exchange() 必须返回剩余的点数值，方法若会返回值，必须在方法前声明返回值的类型❸。

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示 »» 在 Java 命名习惯中，方法名称首字母是小写。

Lab

其实如果是直接建立三个 CashCard 对象，然后进行储值并显示明细，可以这样使用数组，让程序更简洁：

Encapsulation2 CashApp.java

```
package cc.openhome;

import java.util.Scanner;

public class CardApp {
    public static void main(String[] args) {
        CashCard[] cards = {
            new CashCard("A001", 500, 0),
            new CashCard("A002", 300, 0),
            new CashCard("A003", 1000, 1)
        };

        Scanner scanner = new Scanner(System.in);
        for(CashCard card : cards) {
            System.out.printf("为 (%s, %d, %d) 储值：",card.number, card.balance,
card.bonus);
            card.store(scanner.nextInt());
            System.out.printf("明细 (%s, %d, %d)%n",card.number, card.balance,
card.bonus);
        }
    }
}
```

执行结果如下所示：

```
为 (A001, 500, 0) 储值：1000
明细 (A001, 1500, 1)
为 (A002, 300, 0) 储值：2000
明细 (A002, 2300, 1)
为 (A003, 1000, 1) 储值：3000
明细 (A003, 4000, 2)
```

5.1.3 封装对象内部数据

在前一个范例中，CashCard 类上定义了 store() 等方法，这样才可以执行 store() 等方法中的相关条件检查流程：

```
CashCard card1 = new CashCard("A001", 500, 0);
card1.store(scanner.nextInt());
```

CashCard 用户还是可以像下面这样撰写程序，跳过相关条件检查：

```
CashCard card1 = new CashCard("A001", 500, 0);
card1.balance += scanner.nextInt();
card1.bonus += 100;
```

Lab

问题在哪儿？因为你没有封装 CashCard 中不想让其他用户直接存取的私有数据，其他用户撰写程序时，就有了自由存取类私有数据的选择，如果有些数据是类所私有，在 Java 中可以使用 **private** 关键字定义：

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Encapsulation3 CashCard.java

```
package cc.openhome;
class CashCard {
    private String number;
    private int balance;
    private int bonus;
    ...
    void store(int money) {
        if(money > 0) {
            this.balance += money;
            if(money >= 1000) {
                this.bonus++;
            }
        }
        else {
            System.out.println("储值是负的？你是来乱的吗？");
        }
    }

    int getBalance() {
        return balance;
    }

    int getBonus() {
        return bonus;
    }

    String getNumber() {
        return number;
    }
}
```

❶ 使用 private 定义私有成员

❷ 要修改 balance，得通过 store() 定义的流程

❸ 提供取值方法成员

在这个例子，不想让其他用户直接存取 number、balance 与 bonus，所以使用 private 声明❶，这样一来，编译程序会使其他用户在直接存取 number、balance 与 bonus 时编译失败，如图 5.1 所示。

如果没有提供方法存取 private 成员，那其他用户就不能存取。在 CashCard 的例子中，如果想修改 balance 或 bonus，就一定得通过 store()、charge()、exchange()等方法，也就一定得经过你定义的流程❷。

如果没办法直接取得 number、balance 与 bonus，那这段程序代码怎么办？如图 5.2 所示。

```
Scanner user
CashCard car
card1.balance += userInput.nextInt();
card1.bonus += 100;
```

balance has private access in CashCard

(Alt-Enter shows hints)

图 5.1 不能存取 private 成员

```
System.out.prin
        card1.number, card1.balance, card1.bonus);
System.out.printf("明細 (%s, %d, %d)%n",
        card2.number, card2.balance, card2.bonus);
System.out.printf("明細 (%s, %d, %d)%n",
        card3.number, card3.balance, card3.bonus);
```

number has private access in CashCard

(Alt-Enter shows hints)

图 5.2 不能存取 private 成员怎么办

除非你愿意提供取值方法(Getter)，让其他用户可以取得 number、balance 与 bonus 的值，否则其他用户一定无法取得。基于你的意愿，CashCard 类上定义了 getNumber()、getBalance()与 getBonus()等取值方法❸，所以可以这样修改程序：

严禁网络传播本PDF，违者责任自负！

```
System.out.printf("明细 (%s, %d, %d)%n",
            card1.getNumber(), card1.getBalance(), card1.getBonus());
System.out.printf("明细 (%s, %d, %d)%n",
            card2.getNumber(), card2.getBalance(), card2.getBonus());
System.out.printf("明细 (%s, %d, %d)%n",
            card3.getNumber(), card3.getBalance(), card3.getBonus());
```

在 Java 命名规范中，取值方法的名称形式是固定的，也就是以 get 开头，之后接上首字母大写的单词。在 IDE 中，可以使用程序代码自动产生功能来生成取值方法，以 NetBeans 为例，可以在类原始码中右击，在弹出的快捷菜单中选择“插入代码”|“构造函数”命令，打开“生成构造函数”对话框，可以选择想产生哪些数据成员的取值方法，单击“生成”按钮就可以自动生成取值方法的程序代码，如图 5.3 所示。

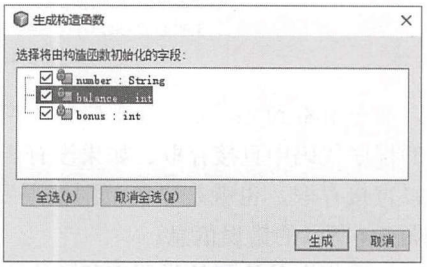


图 5.3 自动生成取值方法

所以你封装了什么？封装了类私有数据，让其他用户无法直接存取，而必须通过你提供的操作方法，经过你定义的流程才有可能存取私有数据。事实上，其他用户也无从得知你的类中有哪些私有数据，所以不会知道对象的内部细节。

在这里对封装做个小小总结，封装目的主要就是隐藏对象细节，将对象当作黑箱进行操作。就如前面的范例，其他用户会调用构造函数，但不知道构造函数的细节；他们会调用方法，但不知道方法的流程；他们也不会知道有哪些私有数据，要操作对象，一律得通过你提供的方法调用。

private 也可以用在方法或构造函数声明上，私有方法或构造函数通常是类内部某个共享的演算流程，外界不用知道私有方法的存在。private 也可以用在内部类声明，内部类会在稍后介绍。

提示 »» 私有构造函数的实际例子会在枚举（enum）时看到，详情可参考第 18 章。

5.2 类语法细节

面向对象概念是抽象的，不同程序语言会用不同语法来支持概念的实现。前一节介绍了面向对象中封装的通用概念以及如何用 Java 语法实现，接下来则要介绍 Java 的特定语法细节。

5.2.1 public 权限修饰

前一节的 CashCard 类是定义在 cc.openhome 包中，假设现在为了管理上的需求，要将 CashCard 类定义至 cc.openhome.virtual 包中，除了原始码与位码的文件夹需求必须符合包层级之外，原始码内容也得做些修改：

```
package cc.openhome.virtual;
class CashCard {
    ...
}
```

修改过后，可以发现，使用到 CashCard 的 CardApp 出错了，根据第 2 章有关 package 与 import

的介绍，因为 CashCard 与 CardApp 不同包，应该在 CardApp 加上 import 描述，可是加上后还是显示图 5.4 所示错误。

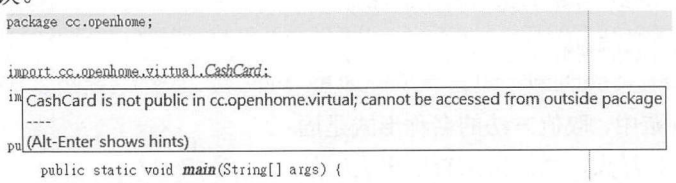


图 5.4 CashCard 不是公开的？

前一节看到 private 权限修饰，声明 private 的成员表示为类私有，其他用户无法在其他类的程序代码中直接存取。如果没有声明权限修饰的成员，只有在相同包的类程序代码中，才可以直接存取，也就是“包范围权限”。如果不同包的类程序代码中，想要直接存取，就会出现图 5.4 所示的错误信息。

如果想在其他包的类程序代码中存取某包的类或对象成员，则该类或对象成员必须是公开成员，在 Java 中要使用 **public** 加以声明。例如：

Public CashCard.java

```
package cc.openhome.virtual;

public class CashCard {                                    ❶ 这是个公开类
    ...
    public CashCard(String number, int balance, int bonus) {   ❷ 这是个公用构造函数
        ...
    }

    public void store(int money) {
        ...
    }

    public void charge(int money) {
        ...
    }

    public int exchange(int bonus) {
        ...
    }                                                      ❸ 这些是公开方法

    public int getBalance() {
        return balance;
    }

    public int getBonus() {
        return bonus;
    }

    public String getNumber() {
        return number;
    }
}
```

可以声明类为 public，这表示它是个公开类，可以在其他包的类中使用❶。在构造函数上声明 public，表示其他包中的类可以直接调用这个构造函数❷。在方法上声明 public，表示其他包的方法中可以直接调用这个方法❸。如果愿意，也可以在对象数据成员上声明 public。

注意>>> 由于 Java SE 9 导入了模块系统，如果你采用模块设计，想要给其他模块存取的包，必须在模块描述文件声明，否则就算是 public 的类或方法等，其他模块也无法存取。

在 2.2.2 节介绍过，包管理其实还有权限管理上的概念，没有定义任何权限关键字时，就是包权限。在 Java 中其实有 private、protected 与 public 三个权限修饰，你已经认识 private 与 public 的使用了，protected 则会在第 6 章说明。

提示>>> 如果类上没有声明 public 权限关键字，类中的方法声明就算是 public，也等于是包权限了，因为类本身是包权限，其他包根本就无法使用类，更别说当中定义的方法。

5.2.2 关于构造函数

在定义类时，可以使用构造函数定义对象建立的初始流程。构造函数是与类名称同名，无须声明返回类型的方法。例如：

```
public class Some {
    private int a = 10;             // 指定初始值
    private String text;            // 默认值 null
    public Some(int a, String text) {
        this.a = a;
        this.text = text;
    }
    ...
}
```

如果这样建立 Some 对象，成员 a 与 text 会初始两次：

```
Some some = new Some(10, "some text");
```

创建对象时，数据成员就会初始化，如果没有指定初始值，则会使用默认值初始化。默认值如表 5.1 所示。

表 5.1 数据成员初始值

数据类型	初始值
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	\u0000
boolean	false
类	null

所以使用 new 创建 Some 对象时，a 与 text 分别先初始为 10 与 null，之后会再通过构造函数流程，设定为构造函数参数的值。如果定义类时，没有撰写任何构造函数，编译程序会自动加入一个无参数、内容为空的构造函数。

正因为编译程序会在你没有撰写任何构造函数时，自动加入默认构造函数(Default Constructor)，所以在没有撰写任何构造函数时，也可以这样以无自变量方式调用构造函数：

```
Some some = new Some();
```

提示 »» 无参数的构造函数也被称为 Nullary 构造函数。只有编译程序自动加入的构造函数，才称为默认构造函数。如果自行撰写无参数、没有内容的构造函数，就不称为默认构造函数了（只是一个 Nullary 构造函数）。虽然只是名词定义，不过认证考试时要区别一下两者的不同。

如果自行撰写了构造函数，编译程序就不会自动建立默认构造函数。如果这么写：

```
public class Some {
    public Some(int a) {
    }
}
```

那就只有一个具 int 参数的构造函数，所以就不可以用 new Some()来创建对象，而必须使用 new Some(1)的形式来创建对象。

5.2.3 构造函数与方法重载

视使用情境或条件的不同，创建对象时也许希望有对应的初始流程。可以定义多个构造函数，只要参数类型或个数不同，这称为重载(Overload)构造函数。例如：

```
public class Some {
    private int a = 10;
    private String text = "n.a.";
    public Some(int a) {
        if(a > 0) {
            this.a = a;
        }
    }
    public Some(int a, String text) {
        if(a > 0) {
            this.a = a;
        }
        if(text != null) {
            this.text = text;
        }
    }
    ...
}
```

在这个代码段中，创建时有两种选择：一是使用 new Some(100)的方式，二是使用 new Some(100, "some text")的方式。

提示 »» 有些场合建议，如果定义了有参数的构造函数，也可以加入无参数的构造函数，即使内容为空也无所谓，这是为了日后使用上的弹性。例如，运用反射(Reflection)机制生成对象的需求，或者继承时调用父类构造函数时的方便。

定义方法时也可以进行重载，可为类似功能的方法提供统一名称，但根据参数类型或个数的不同调用对应的方法。以 String 类为例，其 valueOf()方法就提供了多个版本：

```
public static String valueOf(boolean b)
public static String valueOf(char c)
public static String valueOf(char[] data)
public static String valueOf(char[] data, int offset, int count)
public static String valueOf(double d)
public static String valueOf(float f)
public static String valueOf(int i)
public static String valueOf(long l)
```

```
public static String valueOf(Object obj)
```

虽然调用的方法名称都是 valueOf()，但根据传递的自变量类型不同，会调用对应的方法。例如，调用 String.valueOf(10)，因为 10 是 int 类型，所以会执行 valueOf(int i)的版本；若是 String.valueOf(10.12)，因为 10.12 是 double 类型，会执行 valueOf(double d)的版本(片段中看到的 static 稍后就会说明)。

方法重载让程序设计人员不用苦恼方法名称的设计，可用一致的名称来调用类似功能的方法。方法重载可根据传递自变量的类型不同，也可根据参数列个数的不同来设计方法重载。例如：

```
public class SomeClass {
    public void someMethod() {
    }
    public void someMethod(int i) {
    }
    public void someMethod(float f) {
    }
    public void someMethod(int i, float f) {
    }
}
```

注意>>> 返回值类型不可作为方法重载依据，例如以下方法重载并不正确，编译程序会将两个 someMethod()视为重复定义而编译失败：

```
public class Some {
    public int someMethod(int i) {
      return 0;
    }
    public double someMethod(int i) {
        return 0.0;
    }
}
```

在 JDK5 之后使用方法重载时，要注意自动装箱、拆箱问题，来看看以下程序的结果会是什么。

Class OverloadBoxing.java

```
package cc.openhome;

class Some {
    void someMethod(int i) {
        System.out.println("int 版本被调用");
    }
    void someMethod(Integer integer) {
        System.out.println("Integer 版本被调用");
    }
}

public class OverloadBoxing {
    public static void main(String[] args) {
        Some s = new Some();
        s.someMethod(1);
    }
}
```

结果显示“int 版本被调用”，如果想调用参数为 Integer 版本的方法，要明确指定。例如：

```
s.someMethod(new Integer(1));
```

编译程序在处理重载方法时，会依以下顺序来处理：

(1) 还没有装箱动作前可符合自变量个数与类型的方法。
(2) 装箱动作后可符合自变量个数与类型的方法。
(3) 尝试有不定长度自变量(稍后说明)并可符合自变量类型的方法。
(4) 找不到合适的方法，编译错误。

5.2.4 使用 this

除了被声明为 static 的地方外，this 关键字可以出现在类中任何地方，在对象建立后为“这个对象”的参考名称。目前你看到的应用，就是在构造函数参数与对象数据成员同名时，可用 this 加以区别。

```
public class CashCard {
    private String number;
    private int balance;
    private int bonus;
    public CashCard(String number, int balance, int bonus) {
        this.number = number;              // 参数 number 指定给这个对象的 number
        this.balance = balance;            // 参数 balance 指定给这个对象的 balance

        this.bonus = bonus;                // 参数 bonus 指定给这个对象的 bonus
    }
    ...
}
```

在 5.2.3 节中有这个程序片段：

```
public class Some {
    private int a = 10;
    private String text = "n.a.";
    public Some(int a) {
        if(a > 0) {
            this.a = a;
        }
    }
    public Some(int a, String text) {
        if(a > 0) {
            this.a = a;
        }
        if(text != null) {
            this.text = text;
        }
    }
    ...
}
```

在构造函数部分你发现了什么？粗体字部分流程是重复的，重复在程序设计中是个不好的味道(Bad Smell)，你可以在构造函数中调用另一个已定义的构造函数。例如：

```
public class Some {
    private int a = 10;
    private String text = "n.a.";
    public Some(int a) {
```

```
        if(a > 0) {
            this.a = a;
        }
    }
    public Some(int a, String text) {
        this(a);
        if(text != null) {
            this.text = text;
        }
    }
    ...
}
```

在 Java 中，**this()** 代表了调用另一个构造函数，至于调用哪个构造函数，则视调用 this() 时给的自变量类型与个数而定。在上例中，this(a) 会调用 Some(int a) 版本的构造函数，再执行 if(text != null) 之后的程序代码。

注意»» this() 调用只能出现在构造函数的第一行。

在创建对象之后、调用构造函数之前，若有想执行的流程，可以使用{}定义。直接来看范例比较清楚：

Class ObjectInitialBlock.java

```
package cc.openhome;

class Other {
    {
        System.out.println("对象初始区块");
    }

    Other() {
        System.out.println("Other() 构造函数");
    }

    Other(int o) {
        this();
        System.out.println("Other(int o) 构造函数");
    }
}

public class ObjectInitialBlock {
    public static void main(String[] args) {
        new Other(1);
    }
}
```

在这个范例中，调用了 Other(int o) 版本的构造函数，而其中使用 this() 调用了 Other() 版本的构造函数。如果撰写了对象初始区块，对象建立之后会先执行对象初始区块，接着才调用指定的构造函数，所以结果就是：

```
对象初始区块
Other() 构造函数
Other(int o) 构造函数
```

在 3.1.2 节介绍过 final 关键字，如果局部变量声明了 final，表示设值后就不能再变动，对象数据成员上也可以声明 **final**。如以下程序片段：

```
class Something {
    final int x = 10;
    ...
}
```

同样地，程序中其他地方不能再有对 x 设值的动作，否则会编译错误。那以下的程序片段呢？

```
public class Something {
    final int x;
    ...
}
```

将 x 设为默认初始值 0，而其他地方不能再对 x 设值？不对，如果对象数据成员被声明为 **final**，但没有明确使用=指定值，那表示延迟对象成员值的指定，在构造函数执行流程中，一定要有对该数据成员指定值的动作，否则编译错误，如图 5.5 所示。

在图 5.5 中，虽然 Something(int x)版本的构造函数对 final 对象成员 x 设值，但如果用户调用了 Something()版本的构造函数，那 x 就不会被设值，因而编译错误。如果改为以下代码就可以通过编译：

```
class Something {
    final int x;
    Something() {
        this(10);
    }
    Something(int x) {
        this.x = x;
    }
}
```

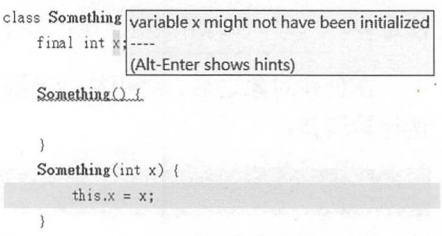


图 5.5 x 有可能没有初始值的编译错误

5.2.5 static 类成员

来看看一个程序片段：

```
class Ball {
    double radius;
    final double PI = 3.14159;
    ...
}
```

如果建立了多个 Ball 对象，那每个 Ball 对象都会有自己的 radius 与 PI 成员，如图 5.6 所示。

不过我们都知道，圆周率其实是个固定的常数，不用每个对象各自拥有，可以在 PI 上声明 **static**，表示它属于类：

```
class Ball {
    double radius;
    static final double PI = 3.141596;
    ...
}
```

被声明为 **static** 的成员，不会让个别对象拥有，而是属于类。如上定义后，如果建立多个 Ball 对象，每个 Ball 对象只会各自拥有 radius，如图 5.7 所示。

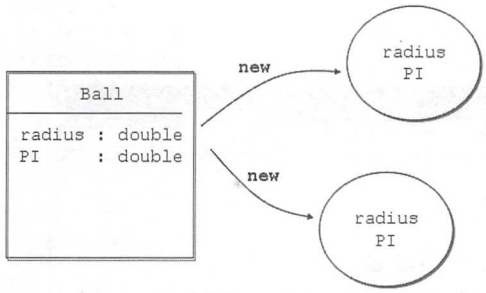


图 5.6 每个 Ball 都拥有自己的 radius 与 PI 数据成员

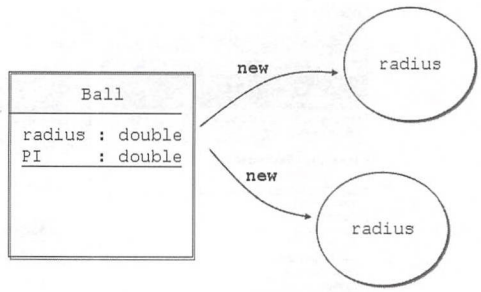


图 5.7 PI 属于 Ball 类拥有

被声明为 static 的成员，是将类名称作为名称空间。也就是说，可以这样取得圆周率：

```
System.out.println(Ball.PI);
```

也就是通过类名称与“.”运算符，就可以取得 static 成员。也可以声明方法为 static 成员。例如：

```
class Ball {
    double radius;
    static final double PI = 3.141596;
    static double toRadians(double angdeg) { // 角度转径度
        return angdeg * (Ball.PI / 180);
    }
}
```

被声明为 static 的方法，也是将类名称作为名称空间，可以通过类名称与“.”运算符来调用 static 方法：

```
System.out.println(Ball.toRadians(100));
```

虽然语法上，也是可以通过参考名称存取 static 成员，但并不建议这样撰写：

```
Ball ball = new Ball();
System.out.println(ball.PI);                  // 极度不建议
System.out.println(ball.toRadians(100));  // 极度不建议
```

Java 程序设计领域，早就有许多良好命名习惯，没有遵守习惯并不是错，但会造成沟通与维护的麻烦。以类命名实例来说，首字母是大写，以 static 使用习惯来说，是通过类名称与“.”运算符来存取。在大家都遵守命名习惯的情况下，看到首字母大写就知道它是类，通过类名称与“.”运算符来存取，就会知道它是 static 成员。比如 System.out、System.in，out 就是 System 拥有的 static 成员，in 也是 System 拥有的 static 成员，这可以查看 API 文件得知，如图 5.8 所示。

继续单击 out 链接就会看到完整声明(有兴趣的读者也可以看 src.zip 中的 System.java)，如图 5.9 所示。

所以 out 实际上是 java.io.PrintStream 类型，被声明为 static，属于 System 类拥有。前面遇到过的例子还有 Integer.parseInt()、Long.parseLong()等剖析方法，根据命名习惯，首字母大写就是类，类名称加上“.”运算符直接调用的就是 static 成员。可以自行查询 API 文件来确认。

正如前面 Ball 类所示范的，static 成员属于类所拥有，将类名称当作名称空间是其最常使用的方式。例如，在 Java SE API 中，只要想到与数学相关的功能，就会想到 java.lang.Math，因为有许多以 Math 类为名称空间的常数与公用方法，如图 5.10 所示。

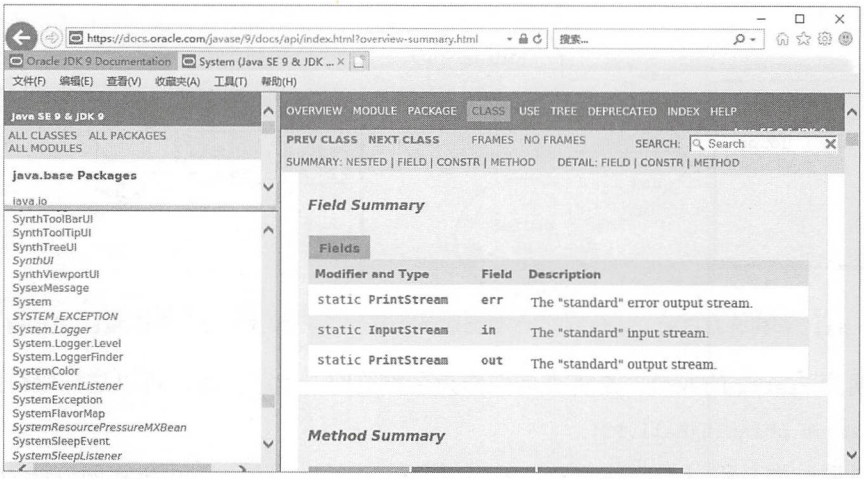


图 5.8 System.err、System.in、System.out 都是 static

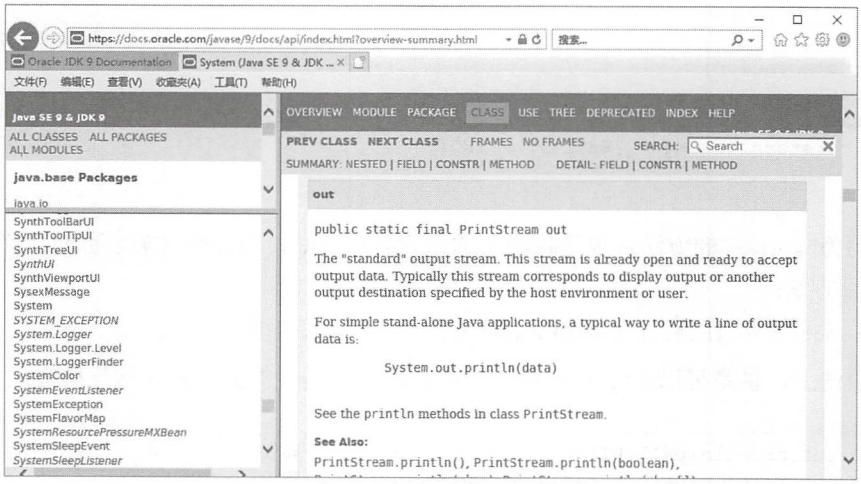


图 5.9 out 的完整声明

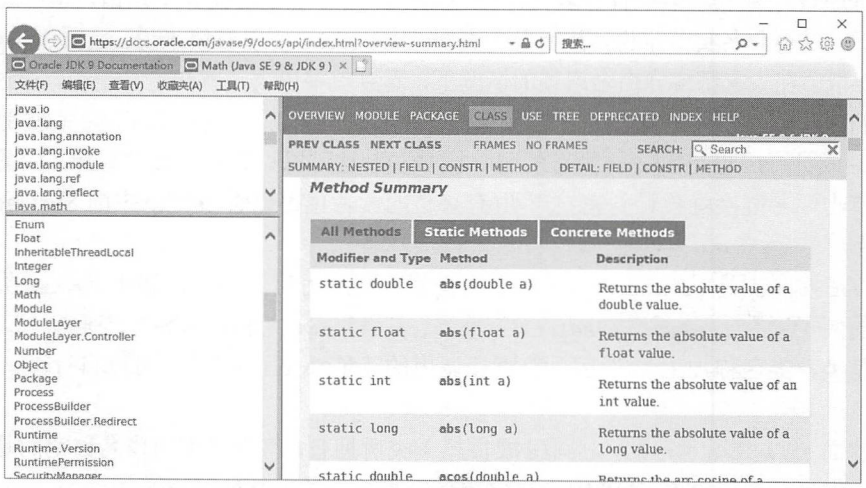


图 5.10 以 java.lang.Math 为名称空间的常数与方法

因为都是 static 成员，所以就可以这么使用：

```
jshell> Math.PI;
$1 ==> 3.141592653589793

jshell> Math.toRadians(100);
$2 ==> 1.7453292519943295
```

由于 static 成员是属于类，而非个别对象，所以在 static 成员中使用 this，会是一种语意上的错误。具体来说，就是在 **static** 方法或区块(稍后说明)中不能出现 **this** 关键字，如图 5.11 所示。

```
class Ball {
    double radius;
    static void do
        double r =this.radius;

    }
```

non-static variable this cannot be referenced from a static context
---
(Alt-Enter shows hints)

图 5.11 static 方法中不能使用 this

如果在程序代码中撰写了某个对象数据成员，虽然没有撰写 this，但也隐含了这个对象某成员的意思，如图 5.12 所示。

```
class Ball {
    double radius;
    static void do
        double r = radius;

    }
```

non-static variable radius cannot be referenced from a static context
---
(Alt-Enter shows hints)

图 5.12 static 方法中不能用非 static 数据成员

在图 5.12 中，虽然撰写 radius，但隐含了 this.radius 的意义，因此会编译错误。static 方法或区块中，也不能调用非 static 方法，如图 5.13 所示。

```
class Ball {
    double radius;

    void doOther() {
    }
    static void d
        doOther();

    }
```

non-static method doOther() cannot be referenced from a static context
---
(Alt-Enter shows hints)

图 5.13 static 方法中不能用非 static 方法成员

在图 5.13 中，虽然撰写 doOther()，但实际隐含了 this.doOther()，因此会编译错误。static 方法或区块中，可以使用 static 数据成员或方法成员。例如：

```
class Ball {
    static final double PI = 3.141596;
    static void doOther() {
        double o = 2 * PI;
    }

    static void doSome() {
        doOther();
    }
    ...
}
```

提示>>> 我个人的习惯是，即使在同一类中，也明确使用类名称加上“.”运算符来调用 static 成员。例如：

```
class Ball {
    static final double PI = 3.141596;
    static void doOther() {
        double o = 2 * Ball.PI;
    }

    static void doSome() {
        Ball.doOther();
    }
    ...
}
```

如果有些动作想在位码加载后执行，则可以定义 static 区块。例如：

```
class Ball {
    static {
        System.out.println("位码加载后就会被执行");
    }
}
```

在这个例子中，Ball.class 加载 JVM 后，默认就会执行 static 区块。

提示>>> 第 16 章会介绍到 JDBC，加载 JDBC 驱动程序的方式之一就是运用 Class.forName()动态加载 Driver 操作类的位码：

```
Class.forName("com.mysql.jdbc.Driver");
```

这个代码段会将 Driver.class 载入 JVM，而 com.mysql.jdbc.Driver 的原始码中，就是在 static 区块中进行驱动程序实例注册的动作：

```
public class Driver extends NonRegisteringDriver
                    implements java.sql.Driver {
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
    ...
}
```

在 JDK5 之后，新增了 import static 语法，可以在使用静态成员时少打几个字。例如：

Class ImportStatic.java

```
package cc.openhome;

import java.util.Scanner;
import static java.lang.System.in;
import static java.lang.System.out;

public class ImportStatic {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(in);
        out.print("请输入姓名: ");
        out.printf("%s 你好! %n", scanner.nextLine());
    }
}
```

原本编译程序看到 in 时，并不知道 in 是什么，但想起你用 import static 告诉过它，想针对 java.lang.System.in 这个 static 成员偷懒，所以就试着用 java.lang.System.in 编译看看，结果就成功了，out 也是同样的道理。

如果一个类中有多个 static 成员想偷懒，也可以使用“*”。例如将上例中 import static 的两行改为如下一行，也可以编译成功：

```
import static java.lang.System.*;
```

提示>>> 后面的程序片段或范例，会适时使用 import static 来简化 static 成员或方法的使用，以增加可读性。

与 import 一样，import static 语法是为了偷懒，但别偷懒过头，要注意名称冲突问题。有些名称冲突，编译程序可通过以下顺序来解析。

- 局部变量覆盖：选用方法中的同名变量、参数、方法名称。
- 成员覆盖：选用类中定义的同名数据成员、方法名称。
- 重载方法比较：使用 import static 的各个静态成员，若有同名冲突，尝试通过重载判断 。

如果编译程序无法判断，则会回报错误。例如，cc.openhome.Util 定义有 static 的 sort()方法，而 java.util.Arrays 也定义有 static 的 sort()方法，以下情况编译就会出错，如图 5.14 所示。

```
import static java.util.Arrays.*;
import static cc.openhome.Util.*;

reference to sort is ambiguous, both method sort(int[]) in cc.openhome.Util and
method sort(int[]) in java.util.Arrays match
----
(Alt-Enter shows hints)

        sort(new int[] {3, 1, 5});
    }
}
```

图 5.14 到底是要哪个 sort()？

5.2.6 不定长度自变量

在调用方法时，若方法的自变量个数事先无法决定该如何处理，例如 System.out.printf()方法无法事先决定自变量个数：

```
System.out.printf("%d", 10);
System.out.printf("%d %d", 10, 20);
System.out.printf("%d %d %d", 10, 20, 30);
```

在 JDK5 之后支持不定长度自变量(Variable-Length Argument)，可以轻松地解决这个问题。直接来看范例：

Class MathTool.java

```
package cc.openhome;

public class MathTool {
    public static int sum(int... numbers) {
        int sum = 0;
        for(int number : numbers) {
            sum += number;
        }
        return sum;
    }
}
```

要使用不定长度自变量，声明参数列时要在类型关键字后加上...，在 sum() 方法通过用增强式 for 循环来取得不定长度自变量中的每个元素，可以这样使用：

```
System.out.println(MathTool.sum(1, 2));
System.out.println(MathTool.sum(1, 2, 3));
System.out.println(MathTool.sum(1, 2, 3, 4));
```

实际上不定长度自变量是编译程序蜜糖，反编译后就可以一窥究竟：int...声明的变量实际上展开为数组，而调用不定长度自变量的客户端，例如 out.println(MathTool.sum(1, 2, 3))，展开后也是变为数组当作自变量传递。这可以从反编译后的程序代码得知：

```
out.println(
    MathTool.sum(new int[] {1, 2, 3})
);
```

提示>>> 因此，就算在 JDK1.4 之前，也可以使用数组来解决自变量个数无法确认的方法调用。

使用不定长度自变量时，方法上声明的不定长度参数必须是参数列最后一个。例如以下是合法声明：

```
public void some(int arg1, int arg2, int... varargs) {
    ...
}
```

以下是不合法声明：

```
public void some(int... varargs, int arg1, int arg2) {
    ...
}
```

使用两个以上不定长度自变量也是不合法的：

```
public void some(int... varargs1, int... varargs2) {
    ...
}
```

如果使用对象的不定长度自变量，声明的方法相同。例如：

```
public void some(Other... others) {
    ...
}
```

5.2.7 内部类

可以在类中再定义类，这称为内部类(Inner Class)。初学者暂时不会使用到内部类，在这里先简单介绍语法，虽然会无聊一些，不过之后章节就会看到相关应用。

内部类可以定义在类区块之中。例如，以下程序片段建立了非静态的内部类：

```
class Some {
    class Other {
    }
}
```

提示>>> 虽然实际应用上很少看到接下来的写法，不过要使用 Some 中的 Other 类，必须先建立 Some 实例。例如：

```
Some s = new Some();
Some.Other o =  s.new Other();
```

内部类也可以使用 public、protected 或 private 声明。例如：

```
class Some {
    private class Other {
    }
}
```

内部类本身可以存取外部类的成员，通常非静态内部类会声明为 private，这类内部类是辅助类中某些操作而设计，外部不用知道内部类的存在。

内部类也可以声明为 static。例如：

```
class Some {
    static class Other {
    }
}
```

一个被声明为 static 的内部类，通常是将外部类当作名称空间。可以这样建立类实例：

```
Some.Other o = new Some.Other();
```

被声明为 static 的内部类，虽然将外部类当作名称空间，但算是个独立类，它可以存取外部类 static 成员，但不可存取外部类非 static 成员，如图 5.15 所示。

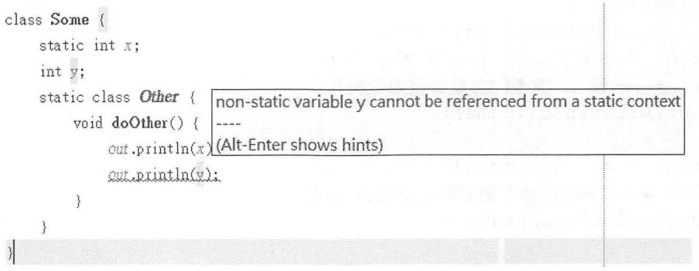


图 5.15 static 内部类不可存取外部类非 static 成员

方法中也可以声明类，这通常是辅助方法中演算之用，方法外无法使用。例如：

```
class Some {
    public void doSome() {
        class Other {
        }
    }
}
```

实际上比较少看到在方法中定义具名的内部类，但经常看到方法中定义匿名内部类(Anonymous Inner Class)并直接实例化，这与类继承或接口操作有关。以下先看一下语法，细节留到第 6 章再做介绍：

```
Object o = new Object() {
    public String toString() {
        return "无聊的语法示范而已";
    }
};
```

如果要稍微解释一下，这个语法定义了一个没有名称的类，它继承 Object 类，并重新定义(Override)了 toString()方法，new 表示实例化这个没有名称的类。匿名类语法本身，在某些场合有时有些啰唆，JDK8 提出了 Lambda，有一部分目的就是用来解决匿名类语法啰唆的问题，第 9 章与第 12 章会再讨论。

5.2.8 传值调用

在一些程序语言，像是C++之类，调用方法传递自变量给参数时，可以有传值调用(Call by Value)或传参考调用(Call by Reference)的方式。Java当中只有传值调用。

提示>>> 传值调用简称传值(Pass by Value)，传参考调用简称传参考(Pass by Reference)。

如果没有接触过具有传值调用与传参考调用选项的程序语言，了解传值调用这个名词对学习Java并没有太大意义。如果接触过C++这类可传值与传参考的语言，注意，C++这类语言中“参考”的意义，跟Java中的“参考”并不相同。

在Java中，调用方法传递对象时要注意的事项，是第4章概念的延伸。举例来说，你觉得以下程序执行的显示结果是什么？

Class CallByValue.java

```
package cc.openhome;

public class CallByValue {
    public static void main(String[] args) {
        Customer c1 = new Customer("Justin");
        some(c1); ←—— ❶ c1 参考的对象会被改变吗?
        System.out.println(c1.name);

        Customer c2 = new Customer("Justin");
        other(c2); ←—— ❷ c2 参考的对象不会被改变吗?
        System.out.println(c2.name);
    }

    static void some(Customer c) {
        c.name = "John"; ←—— ❸改变了哪个对象?
    }

    static void other(Customer c) {
        c = new Customer("Bill"); ←—— ❹ c 参考了哪个对象?
    }
}

class Customer {
    String name;
    Customer(String name) {
        this.name = name;
    }
}
```

先来看执行结果：

```
John
Justin
```

在调用some()方法时传入了c1❶，这表示c1参考的对象也让some()方法的参数c参考，可用图5.16所示。

在some()方法中c.name = "John"❸，就是要求将c参考对象的name成员指定为"John"，结果如图5.17所示。

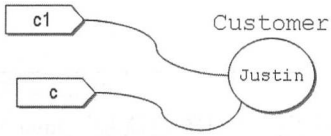


图 5.16 c1 与参数 c 参考同一对象

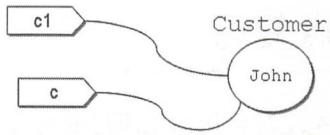


图 5.17 将 c 参考对象的 name 指定为"John"

other()方法执行结束后，c 变量不存在了。下一行取得 c1.name 并显示，从图 5.17 来看，显示 John。

接着看 other()方法调用时传入了 c2❷，这表示 c2 参考的对象也让 other()方法的参数 c 参考，可用图 5.18 所示。

在 other()方法中 c = new Customer("Bill")❹，就是要求建立新对象，并指定给 c 参考，结果如图 5.19 所示。

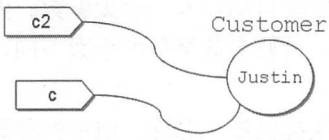


图 5.18 c2 与参数 c 参考同一对象

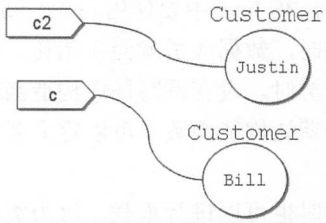


图 5.19 c 参考至新建的对象

some()方法执行结束后，c 变量不存在了，下一章会介绍到，原本 c 参考的对象会被 JVM 清除。下一行取得 c2.name 并显示，从图 5.19 来看，显示 Justin。

这样的行为就是传值，以上示范的是传递对象给参数的情况。如果由方法中返回对象，并指定给变量，也是这种行为。例如：

```
public Customer create (String name) {
    Customer c = new Customer(name);
    ...
    return c;
}
public void doService() {
    Customer customer = create("Irene");
    ...
}
```

在这个程序片段中，doService()方法中调用了 create()方法，在 create()方法结束后，customer 参考的对象与 create()执行过程中 c 参考的对象是同一对象。

这样的行为确实就是传值，而不是传参考。再次强调，Java 中的“参考”与 C++之类语言中的“参考”，在根本上就是不同的定义，只不过刚好都叫作“参考”罢了。

提示»» 偶尔遇到一些学过 C++的学员，总会争辩 Java 中这种行为就是传参考，建议先搞清楚 C++中到底何谓传值与传参考：

http://openhome.cc/Gossip/CppGossip/PassBy.html

http://openhome.cc/Gossip/CppGossip/returnBy.html

然后再看看有关 Java 对象在底层内存的做法：

http://openhome.cc/Gossip/JavaEssence/CallByValue.html

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

5.3 重点复习

构造函数实现对象初始化流程的封装。方法封装了操作对象的流程。Java 中还可以使用 private 封装对象私有数据成员。封装的目的主要就是隐藏对象细节，将对象当作黑箱进行操作。

在 Java 命名规范中，取值方法的名称形式是固定的，也就是以 get 开头，之后接上首字母大写的单词。

由于 Java SE 9 导入了模块系统，如果采用模块设计，想要给其他模块存取的包，必须在模块描述文件声明，否则就算是 public 的类或方法等，其他模块也无法存取。

如果没有声明权限修饰的成员，只有在相同包的类程序代码中才可以直接存取，也就是“包范围权限”。如果想在其他包的类程序代码中存取某包的类或对象成员，则该类或对象成员必须是公开成员，在 Java 中要使用 public 加以声明。

创建对象时，数据成员就会初始化，如果没有指定初始值，则会使用默认值初始化。

如果定义类时，没有撰写任何构造函数，编译程序会自动加入一个无参数、内容为空的构造函数，称为默认构造函数。可以定义多个构造函数，只要参数类型或个数不同，这称为重载构造函数。

定义方法时也可以进行重载，可为类似功能的方法提供统一名称，但根据参数类型或个数的不同调用对应的方法。

编译程序在处理重载方法时，会按以下顺序来处理：

(1) 还没有装箱动作前可符合自变量个数与类型的方法。

(2) 装箱动作后可符合自变量个数与类型的方法。

(3) 尝试有不定长度自变量并可符合自变量类型的方法。

(4) 找不到合适的方法，编译错误。

除了被声明为 static 的地方外，this 关键字可以出现在类中任何地方，在对象建立后为“这个对象”的参考名称。this() 代表了调用另一个构造函数，至于调用哪个构造函数，则视调用 this() 时给的自变量类型与个数而定。

如果对象数据成员被声明为 final，但没有明确使用=指定值，表示延迟对象成员值的指定，在构造函数执行流程中，一定要有对该数据成员指定值的动作，否则编译错误。

被声明为 static 的成员，不会让个别对象拥有，而是属于类。

Java 程序设计领域，早就有许多良好命名习惯，没有遵守习惯并不是错，但会造成沟通与维护的麻烦。以类命名实例来说，首字母是大写，以 static 使用习惯来说，是通过类名称与“.”运算符来存取。在大家都遵守命名习惯的情况下，看到首字母大写就知道它是类，通过类名称与“.”运算符来存取，就会知道它是 static 成员。

在 static 方法或区块中不能出现 this 关键字。static 方法中不能用非 static 数据或方法成员。

import static 语法是为了偷懒，但别偷懒过头，要注意名称冲突问题，有些名称冲突编译程序可通过以下顺序来解析。

- 局部变量覆盖：选用方法中的同名变量、参数、方法名称。
- 成员覆盖：选用类中定义的同名数据成员、方法名称。
- 重载方法比较：使用 import static 的各个静态成员，若有同名冲突，尝试通过重载判断。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

在 JDK5 之后支持不定长度自变量，为编译程序蜜糖，展开后变为数组。使用不定长度自变量时，方法上声明的不定长度参数必须是参数列最后一个。使用两个以上不定长度自变量也是不合法的。

5.4 课后练习

1. 据说古代有座波罗教塔由 3 支钻石棒支撑，神在第一根棒上放置 64 个由小到大排列的金盘，命令僧侣将所有金盘从第一根棒移至第三根棒，搬运过程遵守大盘在小盘下的原则，若每日仅搬一盘，在盘子全数搬至第三根棒，此塔将毁损。请撰写程序，可输入任意盘数，根据以上搬运原则显示搬运过程。

2. 如果有个二维数组代表迷宫如下，0 表示道路，2 表示墙壁：

```
int[][] maze = {
    {2, 2, 2, 2, 2, 2, 2},
    {0, 0, 0, 0, 0, 0, 2},
    {2, 0, 2, 0, 2, 0, 2},
    {2, 0, 0, 2, 0, 2, 2},
    {2, 2, 0, 2, 0, 2, 2},
    {2, 0, 0, 0, 0, 0, 2},
    {2, 2, 2, 2, 2, 0, 2}
};
```

假设老鼠会从索引(1, 0)开始，请使用程序找出老鼠如何跑至索引(6, 5)位置，并以■代表墙，◇代表老鼠，显示出走迷宫路径，如图 5.20 所示。

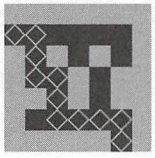


图 5.20 老鼠走迷宫

3. 有个 8 乘 8 棋盘，骑士走法为西洋棋走法，请撰写程序，可指定骑士从棋盘任一位置出发，以标号显示走完所有位置。例如其中一个走法：

```
52 21 64 47 50 23 40  3
63 46 51 22 55  2 49 24
20 53 62 59 48 41  4 39
61 58 45 54  1 56 25 30
44 19 60 57 42 29 38  5
13 16 43 34 37  8 31 26
18 35 14 11 28 33  6  9
15 12 17 36  7 10 27 32
```

4. 国际象棋中皇后可直线前进，吃掉遇到的棋子，如果棋盘上有 8 个皇后，请撰写程序，显示 8 个皇后相安无事地放置在棋盘上的所有方式。例如，其中一个放法如图 5.21 所示。

图 5.21 8 个皇后问题

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 6

继承与多态

学习目标

- 了解继承的目的
- 了解继承与多态的关系
- 知道如何重新定义方法
- 认识 java.lang.Object
- 了解垃圾收集机制

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

6.1 何谓继承

面向对象中，子类继承(Inherit)父类，避免重复的行为定义。不过并非为了避免重复定义行为就使用继承，滥用继承而导致程序维护上的问题时有所闻。如何正确判断使用继承的时机，以及继承之后如何活用多态，才是学习继承时的重点。

6.1.1 继承共同行为

继承基本上就是避免多个类间重复定义共同行为。以实际的例子来说明比较清楚，假设你正在开发一款 RPG(Role-Playing Game)游戏，一开始设定的角色有剑士与魔法师。首先你定义了剑士类：

```
public class SwordsMan {
    private String name;           // 角色名称
    private int level;             // 角色等级
    private int blood;             // 角色血量

    public void fight() {
        System.out.println("挥剑攻击");
    }

    public int getBlood() {
        return blood;
    }
    public void setBlood(int blood) {
        this.blood = blood;
    }

    public int getLevel() {
        return level;
    }
    public void setLevel(int level) {
        this.level = level;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

接着你为魔法师定义类：

```
public class Magician {
    private String name;           // 角色名称
    private int level;             // 角色等级
    private int blood;             // 角色血量

    public void fight() {
        System.out.println("魔法攻击");
    }
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    public void cure() {
        System.out.println("魔法治疗");
    }

    public int getBlood() {
        return blood;
    }
    public void setBlood(int blood) {
        this.blood = blood;
    }

        public int getLevel() {
            return level;
        }
        public void setLevel(int level) {
            this.level = level;
            }

            public String getName() {
                return name;
        }
        public void setName(String name) {
            this.name = name;
        }
}
```

只要是游戏中的角色，都会具有角色名称、等级与血量，类中也都为名称、等级与血量定义了取值方法与设值方法，Magician 中粗体字部分与 SwordsMan 中相对应的程序代码重复了。重复在程序设计上，就是不好的信号。举个例子来说，如果要将 name、level、blood 改为其他名称，那就要修改 SwordsMan 与 Magician 两个类，如果有更多类具有重复的程序代码，那就要修改更多类，造成维护上的不便。

如果要改进，就可以把相同的程序代码提升(Pull Up)为父类。如：

Game1 Role.java

```
package cc.openhome;

public class Role {
    private String name;
    private int level;
    private int blood;

    public int getBlood() {
        return blood;
    }

    public void setBlood(int blood) {
        this.blood = blood;
    }
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

这个类在定义上没什么特别的新语法，只不过是将 SwordsMan 与 Magician 中重复的程序代码复制过来。接着 SwordsMan 可以如下继承 Role：

Game1 SwordsMan.java

```
package cc.openhome;

public class SwordsMan extends Role {
    public void fight() {
        System.out.println("挥剑攻击");
    }
}
```

在这里看到了新的关键字 **extends**，这表示 SwordsMan 会扩充 Role 的行为，也就是继承 Role 的行为，再扩充 Role 原本没有的 fight() 行为。从程序面上来说，Role 中有定义的程序代码，SwordsMan 中都继承且拥有了，并定义了 fight() 方法的程序代码。类似地，Magician 也可以如下定义继承 Role 类。

Game1 Magician.java

```
package cc.openhome;

public class Magician extends Role {
    public void fight() {
        System.out.println("魔法攻击");
    }

    public void cure() {
        System.out.println("魔法治疗");
    }
}
```

Magician 继承 Role 的行为，再扩充了 Role 原本没有的 fight() 与 cure() 行为。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示 »» 在图 6.1 所示的类图中，第一格中的 Role 表示类名称；第二格中的 name、level、blood 表示数据成员；:号之后为各成员类型，-号表示 private；第三格表示方法名称，+号表示 public，:号之后表示返回类型，继承则以空心箭头表示。

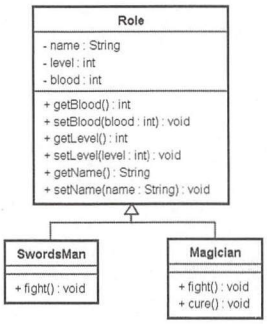


图 6.1 类图

Lab 如何看出确实有继承了呢？从以下简单的程序可以看出。

Game1 RPG.java

```
package cc.openhome;

    public class RPG {
        public static void main(String[] args) {
            demoSwordsMan();
            demoMagician();
        }

        static void demoSwordsMan() {
            SwordsMan swordsMan = new SwordsMan();
            swordsMan.setName("Justin");
            swordsMan.setLevel(1);
            swordsMan.setBlood(200);
            System.out.printf("剑士: (%s, %d, %d)%n", swordsMan.getName(),
                    swordsMan.getLevel(), swordsMan.getBlood());
        }

        static void demoMagician() {
            Magician magician = new Magician();
            magician.setName("Monica");
            magician.setLevel(1);
            magician.setBlood(100);
            System.out.printf("魔法师: (%s, %d, %d)%n", magician.getName(),
                    magician.getLevel(), magician.getBlood());
        }
}
```

虽然 SwordsMan 与 Magician 并没有定义 getName()、getLevel()与 getBlood()等方法，但从 Role 继承了这些方法，所以可以直接使用。执行的结果如下：

```
剑士: (Justin, 1, 200)
魔法师: (Monica, 1, 100)
```

继承的好处之一，就是若要将 name、level、blood 改为其他名称，只要修改 Role.java 就可

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

以了。只要是继承 Role 的子类都无须修改。

注意»»» private 成员无法继承，是不正确的。如果 private 成员无法继承，那为什么上面的范例 name、level、blood 记录的值会显示出来呢？private 成员会被继承，只不过子类无法直接存取，必须通过父类提供的方法来存取(如果父类愿意提供访问方法的话)。

6.1.2 多态与 is-a

在 Java 中，子类只能继承一个父类，继承除了可避免类间重复的行为定义外，还有个重要的关系，那就是子类与父类间会有 is-a 的关系，中文称之为“是一种”的关系，这是什么意思？以前面范例来说，SwordsMan 继承了 Role，所以 **SwordsMan 是一种 Role(SwordsMan is a Role)**，Magician 继承了 Role，所以 **Magician 是一种 Role(Magician is a Role)**。

为何要知道继承时，父类与子类间会有“是一种”的关系？因为要开始理解多态(Polymorphism)，必须先知道操作的对象是“哪一种”东西。

来看实际的例子，以下的代码段，相信你现在可以没有问题地看懂，而且知道可以通过编译：

```
SwordsMan swordsMan = new SwordsMan();
Magician magician = new Magician();
```

那你知道以下的程序片段也可以通过编译吗？

```
Role role1 = new SwordsMan();
Role role2 = new Magician();
```

那你知道以下的程序片段为何无法通过编译呢？

```
SwordsMan swordsMan = new Role();
Magician magician = new Role();
```

编译程序就是语法检查器，要知道以上程序片段为何可以通过编译，为何无法通过编译，就是将自己当作编译程序，检查语法的逻辑是否正确，方式是从=号右边往左读：右边是不是一种左边呢(右边类是不是左边类的子类)？如图 6.2 所示。

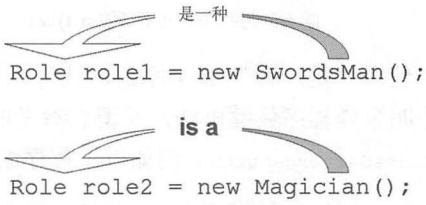


图 6.2 运用 is a 关系判断语法正确性

从右往左读，SwordsMan 是不是一种 Role 呢？是的，所以编译通过。Magician 是不是一种 Role 呢？是的，所以编译通过。同样的判断方式，可以知道为何以下编译失败：

```
SwordsMan swordsMan = new Role();    // Role 是不是一种 SwordsMan?
Magician magician = new Role();      // Role 是不是一种 Magician?
```

编译程序认为第一行 Role 不一定是一种 SwordsMan，所以编译失败；对于第二行，编译程序认为 Role 不一定是一种 Magician，所以编译失败。继续把自己当成编译程序，来看看以下的程序片段是否可以通过编译。

```
Role role1 = new SwordsMan();
SwordsMan swordsMan = role1;
```

这个程序片段最后会编译失败，先从第一行看，SwordsMan 是一种 Role，所以这行可以通过编译。编译程序检查这类语法，一次只看一行，就第二行而言，编译程序看到 role1 为 Role

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

声明的名称，于是检查 Role 是不是一种 SwordsMan，答案是不一定，所以编译失败在第二行。

编译程序会检查父子类间的“是一种”关系，如果你不想要编译程序啰唆，可以叫它住嘴：

```
Role role1 = new SwordsMan();
SwordsMan swordsMan = (SwordsMan) role1;
```

对于第二行，原本编译程序想啰唆地告诉你，Role 不一定是一种 SwordsMan，但你加上了(SwordsMan)让它住嘴了，因为这表示，你就是要让 Role 扮演(Cast)SwordsMan，既然你都明确要求编译程序别啰唆了，编译程序就让这段程序代码通过编译了，不过后果得自行负责。

以上面这个程序片段来说，role1 确实参考至 SwordsMan 实例，所以在第二行让 SwordsMan 实例扮演 SwordsMan 并没有什么问题，所以执行时期并不会出错，如图 6.3 所示。

以下的程序片段，编译可以成功，但执行时会出错：

```
Role role2 = new Magician();
SwordsMan swordsMan = (SwordsMan) role2;
```

对于第一行，Magician 是一种 Role，可以通过编译，对于第二行，role2 为 Role 类型，编译程序原本认定 Role 不一定是一种 SwordsMan 而想要啰唆，但是你明确告诉编译程序，就是要让 Role 扮演为 SwordsMan，所以编译程序就让你通过编译了，不过后果自负。实际上，role2 参考的是 Magician，你要让魔法师假扮为剑士，这在执行上会是个错误，JVM 会抛出 java.lang.ClassCastException，如图 6.4 所示。

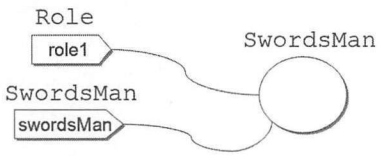


图 6.3 判断是否可扮演(Cast)成功

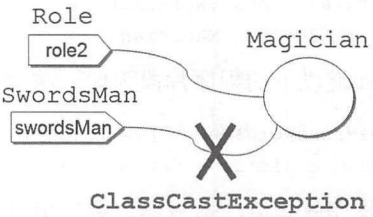


图 6.4 扮演(Cast)失败，执行时抛出 ClassCastException

使用“是一种”(is-a)原则，就可以判断何时编译成功，何时编译失败，以及将扮演(Cast)看作叫编译程序住嘴语法，并留意参考的对象实际类型，就可以判断何时扮演成功，何时会抛出 ClassCastException。例如以下程序编译成功，执行也没问题：

```
SwordsMan swordsMan = new SwordsMan();
Role role = swordsMan;                          // SwordsMan 是一种 Role
```

以下程序片段会编译失败：

```
SwordsMan swordsMan = new SwordsMan();
Role role = swordsMan;                          // SwordsMan 是一种 Role，这行通过编译
SwordsMan swordsMan2 = role;                    // Role 不一定是一种 SwordsMan，编译失败
```

以下程序片段编译成功，执行时也没问题：

```
SwordsMan swordsMan = new SwordsMan();
Role role = swordsMan;                          // SwordsMan 是一种 Role，这行通过编译
// 你告诉编译程序要让 Role 扮演 SwordsMan，以下这行通过编译
SwordsMan swordsMan2 = (SwordsMan) role;        // role 参考 SwordsMan 实例，执行成功
```

以下程序片段编译成功，但执行时抛出 ClassCastException：

```
SwordsMan swordsMan = new SwordsMan();
```


严禁网络传播本PDF，违者责任自负！

```
Role role = swordsMan;                          // SwordsMan 是一种 Role，这行通过编译
// 你告诉编译程序要让 Role 扮演 Magician，以下这行通过编译
Magician magician = (Magician) role;            // role 参考 Magician 实例，执行失败
```

经过以上这一连串的语法测试，好像只是在玩弄语法，不！你懂不懂以上这些东西，将涉及写出来的东西有没有弹性、好不好维护的问题。

有这么严重吗？来出个题目给你吧。请设计 static 方法，显示所有角色的血量。第 5 章刚学过如何定义方法，有的人会撰写以下的方法定义：

```
public static void showBlood(SwordsMan swordsMan) {
    System.out.printf("%s 血量 %d%n",swordsMan.getName(), swordsMan.getBlood());
}
public static void showBlood(Magician magician) {
    System.out.printf("%s 血量 %d%n",magician.getName(), magician.getBlood());
}
```

分别为 SwordsMan 与 Magician 设计 showBlood() 同名方法，这是重载方法的运用，如此就可以如下调用：

```
showBlood(swordsMan);                           // swordsMan 是 SwordsMan 类型
showBlood(magician);                            // magician 是 Magician 类型
```

现在的问题是，目前你的游戏中是只有 SwordsMan 与 Magician 两个角色，如果有 100 个角色呢？重载出 100 个方法？这种方式显然不可能。如果所有角色都是继承自 Role，而且你知道这些角色都是一种 Role，你就可以如下设计方法并调用：

Lab

Game2_RPG.java

```
package cc.openhome;
public class RPG {
    public static void main(String[] args) {
        SwordsMan swordsMan = new SwordsMan();
        swordsMan.setName("Justin");
        swordsMan.setLevel(1);
        swordsMan.setBlood(200);

        Magician magician = new Magician();
        magician.setName("Monica");
        magician.setLevel(1);
        magician.setBlood(100);

        showBlood(swordsMan);  <-- ❶ SwordsMan 是一种 Role
        showBlood(magician);   <-- ❷ Magician 是一种 Role
    }

    static void showBlood(Role role) {  <-- ❸声明为 Role 类型
        System.out.printf("%s 血量 %d%n",role.getName(), role.getBlood());
    }
}
```

在这里仅定义了一个 showBlood() 方法，参数声明为 Role 类型❸。第一次调用 showBlood() 时传入了 SwordsMan 实例，这是合法的语法，因为 SwordsMan 是一种 Role❶。第一次调用 showBlood() 时传入了 Magician 实例也是可行，因为 Magician 是一种 Role❷。执行的结果如下：

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
Justin 血量 200
Monica 血量 100
```

这样的写法好处为何？就算有 100 种角色，只要它们都是继承 Role，都可以使用这个方法显示角色的血量，而不需要像前面重载的方式，为不同角色写 100 个方法，多态的写法显然具有更高的可维护性。

什么叫多态？以抽象讲法解释，就是使用单一接口操作多种类型的对象。若用以上的范例来理解，在 showBlood() 方法中，既可以通过 Role 类型操作 SwordsMan 对象，也可以通过 Role 类型操作 Magician 对象。

提示 »» Java 以继承及界面来实现多态，是次态(Subtype)多态的一种实现，有兴趣的话，可以进一步参考《思考行为外观的次态多态》：

http://openhome.cc/Gossip/Programmer/SubTypePolymorphism.html

注意 »» 稍后会学到 Java 中 interface 的使用，在多态定义中，使用单一接口操作多种类型的对象，这里的接口并不是专指 Java 中的 interface，而是指对象上可操作的方法。

6.1.3 重新定义行为

现在有个需求，请设计 static() 方法，可以播放角色攻击动画。你也许会这么想，学刚刚学过的多态的写法，设计个 drawFight() 方法如何？如图 6.5 所示。

```
static void drawFight(Role role) {
    System.o
    role.fight();
}
```

cannot find symbol
symbol: method fight()
location: variable role of type Role

(Alt-Enter shows hints)

图 6.5 Role 没有定义 fight() 方法

对 drawFight() 方法而言，只知道传进来的会是一种 Role 对象，所以编译程序也只能检查你调用的方法，Role 是不是有定义。很显然，Role 目前并没有定义 fight() 方法，因此编译错误。

然而仔细观察一下 SwordsMan 与 Magician 的 fight() 方法，它们的方法签署(Method Signature)都是：

```
public void fight()
```

也就是说，操作接口是相同的，只是方法操作内容不同。可以将 fight() 方法提升至 Role 类中定义：

Game3 Role.java

```
package cc.openhome;

public class Role {
    ...
    public void fight() {
        // 子类要重新定义 fight() 的实际行为
    }
}
```

在 Role 类中定义了 fight() 方法，由于实际上角色如何攻击，只有子类才知道，所以这里的 fight() 方法内容是空的，没有任何程序代码执行。SwordsMan 继承 Role 之后，再对 fight() 的行为进行定义：

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

Game3 SwordsMan.java

```
package cc.openhome;

public class SwordsMan extends Role {
    public void fight() {
            System.out.println("挥剑攻击");
        }
}
```

在继承父类之后，定义与父类中相同的方法部署，但执行内容不同，这称为重新定义(Override)。因为对父类中已定义的方法执行不满意，所以在子类中重新定义执行。Magician 继承 Role 之后，也重新定义了 fight() 的行为：

Game3 Magician.java

```
package cc.openhome;

public class Magician extends Role {
    public void fight() {
            System.out.println("魔法攻击");
        }
    ...
}
```

由于 Role 定义了 fight() 方法(虽然方法区块中没有程序代码运行)，所以编译程序不会找不到 Role 的 fight()，因此可以如下撰写：

Game3 RPG.java

```
package cc.openhome;

public class RPG {
    public static void main(String[] args) {
        SwordsMan swordsMan = new SwordsMan();
        swordsMan.setName("Justin");
        swordsMan.setLevel(1);
        swordsMan.setBlood(200);

        Magician magician = new Magician();
        magician.setName("Monica");
        magician.setLevel(1);
        magician.setBlood(100);

        drawFight(swordsMan);  <-- ❶ 实际操作的是 SwordsMan 实例
        drawFight(magician);   <-- ❷ 实际操作的是 Magician 实例
}

    static void drawFight(Role role) {  <-- ❸声明为 Role 类型
        System.out.print(role.getName());
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
        role.fight();
    }
}
```

在 drawFight()方法中声明了 Role 类型的参数❸，那方法中调用的到底是 Role 中定义的 fight()，还是个别子类中定义的 fight()呢？如果传入 drawFight()的是 SwordsMan，role 参数参考的就是 SwordsMan 实例，操作的就是 SwordsMan 上的方法定义，如图 6.6 所示。

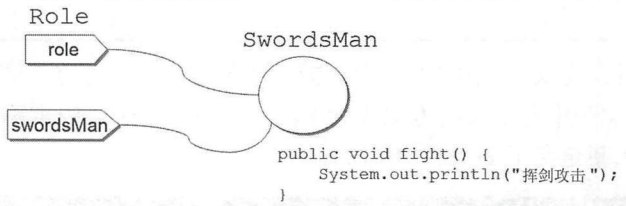


图 6.6　role 牌子挂在 SwordsMan 实例

这就好比 role 牌子挂在 SwordsMan 实例身上，你要求有 role 牌子的对象攻击，发动攻击的对象就是 SwordsMan 实例。同样地，如果传入 drawFight()的是 Magician，role 参数参考的就是 Magician 实例，操作的就是 Magician 上的方法定义，如图 6.7 所示。

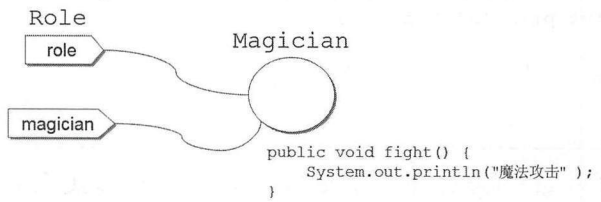


图 6.7　role 牌子挂在 Magician 实例

所以范例最后的执行结果是：

```
Justin 挥剑攻击
Monica 魔法攻击
```

在重新定义父类中某个方法时，子类必须撰写与父类方法中相同的签署，然而如果打错字了：

```
public class SwordsMan extends Role {
    public void Fight() {
        System.out.println("挥剑攻击");
    }
}
```

以这里的例子来说，父类中定义的是 fight()，但子类中定义了 Fight()，这就不是重新定义 fight()了，而是子类新定义了一个 Fight()方法。这是合法的方法定义，编译程序并不会发出任何错误信息，但能在运行范例时，发现为什么 SwordsMan 完全没有攻击。

在 JDK5 之后支持标注(Annotation)，其中一个内建的标准标注就是**@Override**。如果在子类中某个方法前标注@Override，表示要求编译程序检查，该方法是不是真的重新定义了父类中某个方法，如果不是的话，就会引发编译错误，如图 6.8 所示。

如果要重新定义某方法，加上@Override，就不用担心打错字的问题了。关于标注详细语法，会在第 19 章说明。

严禁网络传播本PDF，违者责任自负！　　非卖品，仅供非商业用途或交流学习使用

```
public class SwordsMan extends Role {
        method does not override or implement a method from a supertype
        ----
        (Alt-Enter shows hints)
    @Override
    public void Fight() {
        System.out.println("挥剑攻击");
    }
}
```

图 6.8　编译程序检查是否真的重新定义父类某方法

6.1.4　抽象方法、抽象类

上一个范例 Role 类的定义中，fight()方法区块中实际上没有撰写任何程序代码，虽然满足了多态需求，但会引发的问题是，由于没有任何方式强迫或提示子类一定要操作 fight()方法，只能口头或在文件上告知，不过如果有人没有传达到、没有看文件或文件看漏了呢？

如果某方法区块中真的没有任何程序代码操作，可以使用 **abstract** 标示该方法为抽象方法(Abstract Method)，该方法不用撰写{}区块，直接“;”结束即可。例如：

Game4 Role.java

```
package cc.openhome;

public abstract class Role {
    ...
    public abstract void fight();
}
```

类中若有方法没有操作，并且标示为 abstract，表示这个类定义不完整。定义不完整的类就不能用来生成实例，这就好比设计图不完整，不能用来生产成品一样。Java 中规定内含抽象方法的类，一定要在 **class** 前标示 **abstract**，如上例所示，它表示这是一个定义不完整的抽象类(Abstract Class)。如果尝试用抽象类创建实例，就会引发编译错误，如图 6.9 所示。

```
        Role is abstract; cannot be instantiated
        ----
        (Alt-Enter shows hints)
Role role = new Role();
```

图 6.9　不能实例化抽象类

子类如果继承抽象类，对于抽象方法有两种做法，一种做法是继续标示该方法为 abstract(该子类因此也是个抽象类，必须在 class 前标示 abstract)；另一种做法就是操作抽象方法。如果两种做法都没有实施，就会引发编译错误，如图 6.10 所示。

```
public class SwordsMan extends Role {
        SwordsMan is not abstract and does not override abstract method fight() in Role
        ----
        (Alt-Enter shows hints)
```

图 6.10　没有操作抽象方法

严禁网络传播本PDF，违者责任自负！　　非卖品，仅供非商业用途或交流学习使用

6.2 继承语法细节

上一节介绍了继承的基础概念与语法，然而结合 Java 的特性，继承还有许多细节必须明了，如哪些成员可以限定在子类中使用、哪些方法签署算重新定义、Java 中所有对象都是一种 java.lang.Object 等细节，这将在本节中详细说明。

6.2.1 protected 成员

就上一节的 RPG 游戏来说，如果建立了一个角色，想显示角色的细节，则必须这样撰写：

```
SwordsMan swordsMan = new SwordsMan();
...
System.out.printf("剑士 (%s, %d, %d)%n", swordsMan.getName(),
        swordsMan.getLevel(), swordsMan.getBlood());
Magician magician = new Magician();
...
System.out.printf("魔法师 (%s, %d, %d)%n", magician.getName(),
        magician.getLevel(), magician.getBlood());
```

这对使用 SwordsMan 或 Magician 的客户端有点不方便，如果要在 SwordsMan 或 Magician 上定义 toString() 方法，返回角色的字符串描述：

```
public class SwordsMan extends Role {
    ...
    public String toString() {
        return String.format("剑士 (%s, %d, %d)", this.getName(),
                this.getLevel(), this.getBlood());
    }
}

public class Magician extends Role {
    ...
    public String toString() {
        return String.format("魔法师 (%s, %d, %d)", this.getName(),
                this.getLevel(), this.getBlood());
    }
}
```

客户端就可以这样撰写：

```
SwordsMan swordsMan = new SwordsMan();
...
System.out.println(swordsMan.toString());
Magician magician = new Magician();
...
System.out.printf(magician.toString());
```

看来客户端简洁许多。不过你定义的 toString() 在取得名称、等级与血量时不是很方便，因为 Role 中的 name、level 与 blood 被定义为 private，所以无法直接在子类中存取，只能通过 getName()、getLevel()、getBlood() 来取得。

将 Role 中的 name、level 与 blood 定义为 public，这又会完全开放 name、level 与 blood 访问权限，你并不想这么做。如果只想让子类可以直接存取 name、level 与 blood 的话，可以定义它们为 protected：

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

Game5 Role.java

```
package cc.openhome;

public abstract class Role {
    protected String name;
    protected int level;
    protected int blood;
    ...
}
```

Lab

被声明为 protected 的成员，相同包中的类可以直接存取，不同包中的类可以在继承后的子类直接存取。现在 SwordsMan 可以这样定义 toString()：

Game5 SwordsMan.java

```
package cc.openhome;

public class SwordsMan extends Role {
    ...
    public String toString() {
        return String.format("剑士 (%s, %d, %d)", this.name, this.level, this.blood);
    }
}
```

Lab

Magician 也可以这样撰写：

Game5 Magician.java

```
package cc.openhome;

public class Magician extends Role {
    ...
    public String toString() {
        return String.format("魔法师 (%s, %d, %d)", this.name, this.level, this.blood);
    }
}
```

提示 »» 如果方法中没有同名参数，this 可以省略，不过基于程序可读性，多打个 this 会比较清楚。

到这里为止，Java 中 3 个权限关键字你都看到了，也就是 public、protected 与 private。虽然只有 3 个权限关键字，但实际上有 4 个权限范围，因为没有定义权限关键字，默认就是包范围。权限关键字与权限范围的关系，如表 6.1 所示。

表 6.1 权限关键字与范围

关键字	类内部	相同包类	不同包类
public	可存取	可存取	可存取
protected	可存取	可存取	子类可存取
无	可存取	可存取	不可存取
private	可存取	不可存取	不可存取

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

提示 »» 简单来说，依权限小至大来区分，就是 private、无关键字、protected 与 public，设计时要使用哪个权限，是依经验或团队讨论而定。如果一开始不知道使用哪个权限，就先使用 private，以后视需求再放开权限。

别忘了 Java SE 9 支持模块化，如果是采用模块方式设计，public、protected 还会受到模块描述文件设定的权限限制。

6.2.2 重新定义的细节

Lab

在 6.1.3 节已看过何谓重新定义方法与实例，有时候重新定义方法时，并非完全不满意父类中的方法，只是希望在执行父类中方法的前、后做点加工。例如，也许 Role 类中原本就定义了 toString() 方法：

Game6 Role.java

```
package cc.openhome;

public abstract class Role {
    ...
    public String toString() {
            return String.format("(%s, %d, %d)", this.name,this.level, this.blood);
        }
}
```

Lab

在 SwordsMan 子类中重新定义 toString() 的内容时，可以执行 Role 中的 toString() 方法取得字符串结果，再连接“剑士”字样。在 Java 中，如果想取得父类中的方法定义，可以在调用方法前，加上 **super** 关键字。例如：

Game6 SwordsMan.java

```
package cc.openhome;

public class SwordsMan extends Role {
    ...
    @Override
    public String toString() {
        return "剑士 " + super.toString();
    }
}
```

Lab

类似地，Magician 在重新定义 toString() 时，也可以如法炮制：

Game6 Magician.java

```
package cc.openhome;

public class Magician extends Role {
    ...
    @Override
    public String toString() {
        return "魔法师 " + super.toString();
    }
}
```

非卖品，仅供非商业用途或交流学习使用

可以使用 super 关键字调用的父类方法，不能定义为 private(因为这就限定只能在类内使用)。

重新定义方法要注意，对于父类中的方法权限，只能扩大不能缩小。若原来成员 public，子类中重新定义时不可为 private 或 protected，如图 6.11 所示。

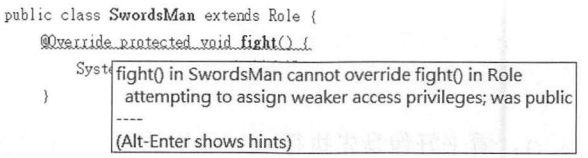


图 6.11　重新定义时不能缩小方法权限

在 JDK5 之前，重新定义方法时除了可以定义权限较大的关键字外，其他部分必须与父类中方法签署的完全一致。例如，我们设计了一个 Bird 类：

```
public class Bird {
    protected String name;
    public Bird(String name) {
        this.name = name;
    }
    public Bird copy() {
        return new Bird(name);
    }
}
```

原先 copy()返回了 Bird 类型，如果 Chicken 继承 Bird，打算让 copy()方法返回 Chicken，那么在 JDK5 之前会发生编译错误，如图 6.12 所示。

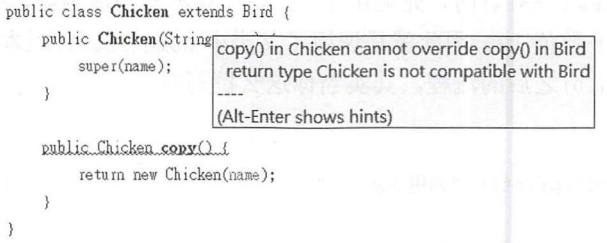


图 6.12　JDK5 之前重新定义方法时，返回类型也必须一致

在 JDK5 之后，重新定义方法时，如果返回类型是父类中方法返回类型的子类，也是可以通过编译的。图 6.12 所示的例子，在 JDK5 中不会出现编译错误。

提示 »» static 方法属于类拥有，如果子类中定义了相同签署的 static 成员，该成员属于子类所有，而非重新定义，static 方法也没有多态，因为对象不会个别拥有 static 成员。

6.2.3　再看构造函数

如果类有继承关系，在创建子类实例后，会先进行父类定义的初始流程，再进行子类定义的初始流程，也就是创建子类实例后，会先执行父类构造函数定义的流程，再执行子类构造函数定义的流程。

构造函数可以重载，父类中可重载多个构造函数，如果子类构造函数中没有指定执行父类中哪个构造函数，默认会调用父类中无参数构造函数。如果这样撰写程序：

```
class Some {
    Some() {
        System.out.println("调用 Some()");
    }
}
class Other extends Some {
    Other() {
        System.out.println("调用 Other()");
    }
}
```

如果尝试 new Other()，看来好像是先执行 Some()中的流程，再执行 Other()中的流程，也就是先显示"调用 Some()"，再显示"调用 Other()"。很奇怪是吧！先继续往下看，就知道为什么了。如果想执行父类中某构造函数，可以使用 **super()**指定。例如：

```
class Some {
    Some() {
        System.out.println("调用 Some()");
    }
    Some(int i) {
        System.out.println("调用 Some(int i)");
    }
}
class Other extends Some {
    Other() {
        super(10);
        System.out.println("调用 Other()");
    }
}
```

在这个例子中，new Other()时，先调用了 Other()版本的构造函数，super(10)表示调用父类构造函数时传入 int 数值 10，因此就是调用了父类中 Some(int i)版本的构造函数，而后再继续 Other()中 super(10)之后的流程。其实当你这么撰写时：

```
class Some {
    Some() {
        System.out.println("调用 Some()");
    }
}
class Other extends Some {
    Other() {
        System.out.println("调用 Other()");
    }
}
```

前面介绍过，如果子类构造函数中没有指定执行父类中哪个构造函数，默认会调用父类中无参数构造函数，也就是等于你这么撰写：

```
class Some {
    Some() {
        System.out.println("调用 Some()");
    }
}
class Other extends Some {
    Other() {
        super();
        System.out.println("调用 Other()");
    }
}
```

所以执行 new Other()时，是先执行 Other()中的流程，而 Other()中指定调用父类无参数构造函数，而后再执行 super()之后的流程。

注意》》 this()与 super()只能择一调用，而且一定要在构造函数第一行执行。

那么你知道图 6.13 为什么会编译错误吗？

在 5.2.2 节介绍过，编译程序会在你没有撰写任何构造函数时，自动加入没有参数的默认构造函数(Default Constructor)，如果自行定义了构造函数，就不会自动加入任何构造函数。在图 6.13 中，Some 定义了有参数的构造函数，所以编译程序不会再加入默认构造函数，Other 的构造函数中没有指定调用父类中哪个构造函数，那就是默认调用父类中无参数构造函数，但父类中现在哪来的无参数构造函数呢？因此编译失败了。

```
class Some {
    Some(int i) {
        out.println("呼叫 Some(int i)");
    }
}

class Other {
    Other() {
        out.println("呼叫 Other()");
    }
}
```

constructor Some in class Some cannot be applied to given types;
required: int
found: no arguments
reason: actual and formal argument lists differ in length
(Alt-Enter shows hints)

图 6.13 找不到构造函数？

提示》》 5.2.3 节提示过一次，有些场合建议，如果定义了有参数的构造函数，也可以加入无参数构造函数，即使内容为空也无所谓，这是为了日后使用上的弹性。例如，运用反射(Reflection)机制生成对象的需求，或者继承时调用父类构造函数时的方便。

6.2.4 再看 final 关键字

在 3.1.2 节中介绍过，如果在指定变量值之后，就不想再改变变量值，可以在声明变量时加上 final 限定，如果后续撰写程序时，用户自己或别人想修改 final 变量，就会出现编译错误。

在 5.2.4 节中也介绍过，如果对象数据成员被声明为 final，但没有明确使用=指定值，那表示延迟对象成员值的指定，在构造函数执行流程中，一定要有对该数据成员指定值的动作，否则编译错误。

class 前也可以加上 final 关键字，如果 **class** 前使用了 **final** 关键字定义，那么表示这个类是最后一个了，不会再有子类，也就是不能被继承。有没有实际的例子呢？有的，String 在定义时就限定为 final 了，这可以在 API 文件上得以验证，如图 6.14 所示。

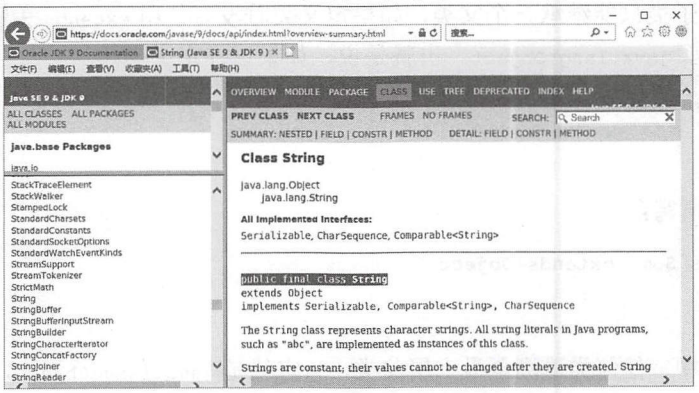


图 6.14 String 是 final 类

如果打算继承 final 类，则会发生编译错误，如图 6.15 所示。

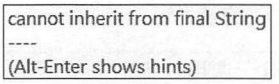


图 6.15　不能继承 final 类

定义方法时，也可以限定该方法为 **final**，这表示最后一次定义方法，也就是子类不可以重新定义 **final** 方法。有没有实际的例子呢？有的，java.lang.Object 上有几个 final 方法，如图 6.16 所示。

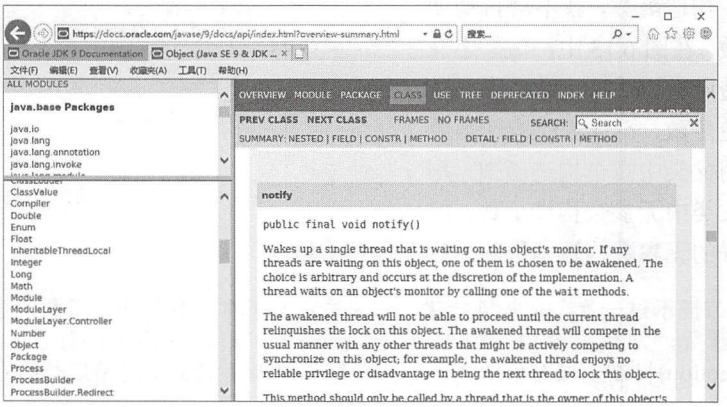


图 6.16　Object 类上的 final 方法之一

如果尝试在继承父类后，重新定义 final 方法，则会发生编译错误，如图 6.17 所示。

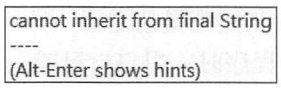


图 6.17　不能重新定义 final 方法

提示 »» 在 Java SE API 中会声明为 final 的类或方法，通常与 JVM 对象或操作系统资源管理有密切相关，因此不希望 API 用户继承或重新定义。

6.2.5 java.lang.Object

在 Java 中，子类只能继承一个父类，如果定义类时没有使用 extends 关键字指定继承任何类，那一定是继承 **java.lang.Object**。也就是说，如果这样定义类：

```
public class Some {
    ...
}
```

那就相当于撰写：

```
public class Some extends Object {
    ...
}
```

因此在 Java 中，任何类追溯至最上层父类，一定就是 java.lang.Object，也就是 Java 中所有对象，一定“是一种”Object，所以这样撰写程序是合法的：

```
Object o1 = "Justin";
Object o2 = new Date();
```

String 是一种 Object，Date 是一种 Object，任何类型的对象都可以使用 Object 声明的名称来参考。这有什么好处？如果有个需求是使用数组收集各种对象，那该声明是什么类型呢？答案是 Object[]。例如：

```
Object[] objs = {"Monica", new Date(), new SwordsMan()};
String name = (String) objs[0];
Date date = (Date) objs[1];
SwordsMan swordsMan = (SwordsMan) objs[2];
```

因为数组长度有限，使用数组来收集对象不是那么方便，以下定义的 ArrayList 类，可以不限长度地收集对象。

Lab

Inheritance_ArrayList.java

```
package cc.openhome;

import java.util.Arrays;

public class ArrayList {
    private Object[] list; ←❶ 使用 Object 数组收集
    private int next; ←❷ 下一个可储存对象的索引

    public ArrayList(int capacity) { ←❸ 指定初始容量
        list = new Object[capacity];
    }

    public ArrayList() {
        this(16); ←❹ 初始容量默认为 16
}

    public void add(Object o) { ←❺ 收集对象方法
        if(next == list.length) { ←❻ 自动增长 Object 数组长度
            list = Arrays.copyOf(list, list.length * 2);
        }
        list[next++] = o;
    }
    public Object get(int index) { ←❼ 依索引取得收集的对象
        return list[index];
    }

    public int size() { ←❽ 已收集的对象个数
        return next;
    }
}
```

自定义的 ArrayList 类，内部使用 Object 数组来收集对象❶，每一次收集的对象会放在 next 指定的索引处❷。在创建 ArrayList 实例时，可以指定内部数组初始容量❸，如果使用无参数构造函数，则默认容量为 16❹。

如果要收集对象，可通过 add() 方法，注意参数的类型为 Object，接收任何对象❺。如果内部数组原长度不够，就使用 Arrays.copyOf() 方法自动建立原长度两倍的数组并复制元素❻。如果想取得收集的对象，可以使用 get() 指定索引取得❼。如果想知道已收集的对象个数，则通过 size() 方法得知❽。

以下使用自定义的 ArrayList 类，可收集访客名称，并将名单转为大写后显示：

Inheritance_Guest.java

```
package cc.openhome;

import java.util.Scanner;
import static java.lang.System.out;

public class Guest {
    public static void main(String[] args) {
        ArrayList names = new ArrayList();
        collectNameTo(names);
        out.println("访客名单：");
        printUpperCase(names);
    }

    static void collectNameTo(ArrayList names) {
        Scanner console = new Scanner(System.in);
        while(true) {
            out.print("访客名称：");
            String name = console.nextLine();
            if(name.equals("quit")) {
                break;
            }
            names.add(name);
        }
    }

    static void printUpperCase(ArrayList names) {
        for(int i = 0; i < names.size(); i++) {
            String name = (String) names.get(i);
            out.println(name.toUpperCase());
        }
    }
}
```

其执行结果如下所示：

```
访客名称：Justin
访客名称：Monica
访客名称：Irene
访客名称：quit
访客名单：
JUSTIN
MONICA
IRENE
```

java.lang.Object 是所有类的顶层父类，这代表了 Object 上定义的方法，所有对象都继承下来了，只要不是被定义为 final 方法，都可以重新定义，如图 6.18 所示。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

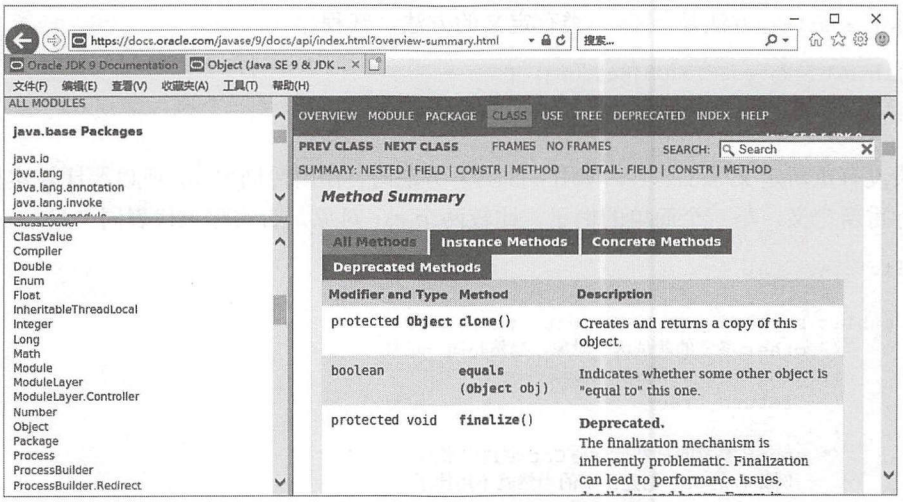


图 6.18 java.lang.Object 定义的方法

1. 重新定义 toString()

举例来说，在 6.2.1 节的范例中，SwordsMan 等类曾定义过 toString()方法，其实 toString()是 Object 上定义的方法。Object 的 toString()默认定义为：

```
public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

目前你不用特别知道这段程序代码的详细内容，总之返回的字符串包括了类名称以及十六进制哈希码，通常这并没有什么阅读上的意义。实际上 6.2.1 节的范例中，SwordsMan 等类是重新定义了 toString()。许多方法若传入对象，默认都会调用 toString()，例如 System.out.print()等方法就会调用 toString()以取得字符串描述来显示，所以 6.2.1 节的这个程序片段：

```
SwordsMan swordsMan = new SwordsMan();
...
System.out.println(swordsMan.toString());
Magician magician = new Magician();
...
System.out.printf(magician.toString());
```

实际上只要这么撰写就可以了：

```
SwordsMan swordsMan = new SwordsMan();
...
System.out.println(swordsMan);
Magician magician = new Magician();
...
System.out.printf(magician);
```

2. 重新定义 equals()

在 4.1.3 节介绍过，在 Java 中要比较两个对象的实质相等性，并不是使用==，而是通过 equals()方法，在后续你看过的 Integer 等打包器以及字符串相等性比较时，使用的都是 equals()方法。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

实际上 equals() 方法是 Object 类有定义的方法，其程序代码是：

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

如果没有重新定义 equals()，使用 equals() 方法时，作用等同于==，所以要比较实质相等性必须自行重新定义。举一个简单的例子，比较两个 Cat 对象是否实际上代表同一只 Cat 的数据：

```
public class Cat {
    ...
    public boolean equals(Object other) {
        // other 参考的就是这个对象，当然是同一对象
        if (this == other) {
            return true;
        }
        /* other 参考的对象是不是 Cat 创建出来的
           例如若是 Dog 创建出来的当然就不用比了 */
        if (!(other instanceof Cat)) {
            return false;
        }
        Cat cat = (Cat) other;
        // 定义如果名称与生日，表示两个对象实质上相等
        if (!getName().equals(cat.getName())) {
            return false;
        }
        if (!getBirthday().equals(cat.getBirthday())) {
            return false;
        }
        return true;
    }
}
```

这个程序片段示范了 equals() 操作的基本概念，相关说明都以批注方式呈现了。这里也看到了 **instanceof** 运算符，它可以用来判断对象是否由某个类创建，左操作数是对象，右操作数是类。在使用 instanceof 时，编译程序还会来帮点忙，会检查左操作数类型是否在右操作数类型的继承架构中或界面操作架构中(第 7 章会说明接口)，如图 6.19 所示。

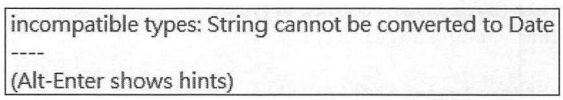


图 6.19 String 与 Date 在继承架构上一点关系也没有

在执行时期，并非只有左操作数对象为右操作数类直接实例化才返回 true，只要左操作数类型是右操作数类型的子类型，instanceof 也是返回 true。

这里仅示范了 equals() 操作的基本概念，实际上操作 equals() 并非这么简单。操作 **equals()** 时通常也会操作 **hashCode()**，原因等到第 9 章学习 Collection 时再说明。如果现在就想知道 equals() 与 hashCode() 操作时要注意的一些事项，可以先参考以下文件：

http://caterpillar.onlyfun.net/Gossip/JavaEssence/ObjectEquality.html

提示 »» 2007 年的研究文献 *Declarative Object Identity Using Relation Types* 中指出，在考察大量 Java 程序代码之后，作者发现大部分 equals() 方法都操作错误。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

6.2.6 关于垃圾收集

创建对象会占据内存，如果程序执行流程中已无法再使用某个对象，该对象就只是徒耗内存的垃圾。

对于不再有用的对象，JVM 有垃圾收集(Garbage Collection，GC)机制，收集到的垃圾对象所占据的内存空间，会被垃圾收集器释放。那么，哪些会被 JVM 认定为垃圾对象？简单地说，在执行流程中，无法通过变量参考的对象，就是 GC 认定的垃圾对象。

什么是执行流程？具体来说就是线程(Thread)(第 11 章才会介绍线程)，目前你唯一接触到的线程就是 `main()`程序进入点开始之后的主线程(也就是主流程)。事实上，关于垃圾收集本身就很复杂，不同的需求也会有不同垃圾收集算法，你只需要知道基本概念即可，细节就交给 JVM 处理。

假设有一个类：

```
public class Some {
    Some next;
}
```

若是从程序进入点开始，有段程序代码如下撰写：

```
Some some1 = new Some();
Some some2 = new Some();
some1 = some2;
```

执行到第二行时，主线程可以通过参考名称所参考到的对象，如图 6.20 所示。

执行到第三行时，是将 `some2` 参考的对象给 `some1` 参考，如图 6.21 所示。

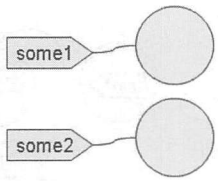


图 6.20 两个对象都有牌子

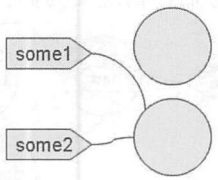


图 6.21 没有牌子的就是垃圾

原先 `some1` 参考的对象不再被任何名称参考，通过主线程也不再能参考到该对象，这个对象就是内存中的垃圾了，GC 会自动找出这些垃圾并予以回收。

GC 的基本概念就是这样，但可以加以变化。如果有段程序是这样：

```
Some some = new Some();
some.next = new Some();
some = null;
```

在执行到第二行时，情况如图 6.22 所示，此时还没有对象是垃圾。

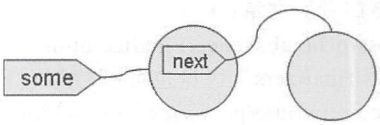


图 6.22 链状参考

由于从主流程开始，可以通过 `some` 参考至中间的对象，而 `some.next` 可以参考至最右边的对象，目前没有必要回收任何对象。执行完成第三行后，情况变成如图 6.23 所示。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

由于从主流程开始，无法通过 some 参考至中间对象，也就无法再通过中间对象的 next 参考至右边对象，所以两个对象都是垃圾。同样的道理，在下面程序代码中，数组参考到的对象全部都会被回收，如图 6.24 所示。

```
Some[] somes = {new Some(), new Some(), new Some()};
somes = null;
```

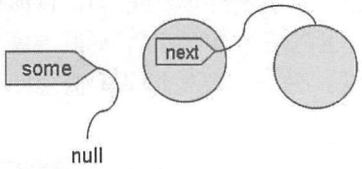


图 6.23　回收几个对象呢？

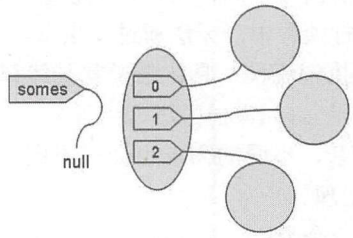


图 6.24　数组参考到的对象被回收

被回收的对象包括数组对象本身，以及三个索引所参考的三个对象。如果是形同孤岛的对象，例如：

```
Some some = new Some();
some.next = new Some();
some.next.next = new Some();
some.next.next.next = some;
some = null;
```

执行到第四行时，情况如图 6.25 所示。

执行完第五行后，情况变为如图 6.26 所示。

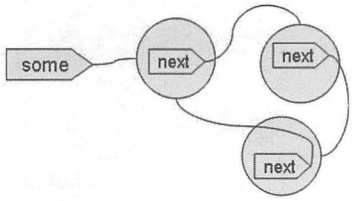


图 6.25　循环参考

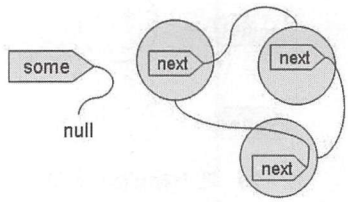


图 6.26　形成孤岛

这个时候形成孤岛的右边三个对象，将全部被 GC 处理掉。

提示 »» GC 在进行回收对象前，会调用对象的 finalize()方法，这是 Object 上就定义的方法。如果在对象被回收前，有些事情想做，可以重新定义 finalize()方法。不过要注意的是，何时启动 GC，要视所采用的 GC 算法而定，也就是 finalize()被调用的时机是无法确定的。

在 *Effective Java* 书中也建议，避免使用 finalize()方法。如果对 finalize()方法有兴趣，可以参考《对象终结者》文章：

http://openhome.cc/Gossip/JavaEssence/Finalize.html

JWorld 上的“避免使用 finalizers”讨论也可以参考一下：

http://www.javaworld.com.tw/jute/post/view?bid=44&id=17264&sty=1&tpg=1&age=0

如果某些对象会大量产生，并在使用之后马上丢弃不用，还得小心效率的问题，定义了 finalize()的对象，会被 JVM 放到一个 finalize 队列，垃圾收集的时间可能被延迟。若大量产生这类对象，可能会因为垃圾收集被持续延后而使得对象迟迟不被回收，最后导致内存不足的错误（OutOfMemoryError）发生。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

6.2.7 再看抽象类

撰写程序常有些看似不合理但又非得完成的需求。举个例子来说，现在老板叫你开发一个猜数字游戏，会随机产生 0~9 的数字，用户输入的数字与随机产生的数字相比，如果相同就显示“猜中了”，如果不同就继续让用户输入数字，直到猜中为止。

这程序有什么难的？相信现在的你也可以写出来：

```
package cc.openhome;

import java.util.Scanner;

public class Guess {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int number = (int) (Math.random() * 10);
        int guess;
        do {
            System.out.print("输入数字：");
            guess = scanner.nextInt();
        } while(guess != number);
        System.out.println("猜中了");
    }
}
```

圆满完成任务了吧？当你将程序交给老板后，老板皱着眉头说：“我有说要在文本模式下执行这个游戏吗？”你就问了：“请问会在哪个环境下执行呢？”老板说：“还没决定，也许会用窗口程序，不过改成网页也不错，唔……下个星期开会讨论一下。”你问：“那可以下星期讨论完我再来写吗？”老板说：“不行！”你(内心 OS)：“当我是哆啦 A 梦喔！我又没有时光机……”

这个例子可笑吗？在团队合作、多个部门开发程序时，有许多时候，你不能只是等另一个部门将程序操作出来，也许另一个部门要 3 个月后才能完成程序操作，难道你们这个部门要空转 3 个月？有些需求无法决定，却要撰写出程序的例子太多了。

有些不合理的需求，本身确实不合理，但有些看似不合理的需求，其实可以通过设计(Design)来解决。以上面的例子来说，取得用户输入、显示结果的环境未定，但你负责的这部分还是可以先操作的。例如：

Inheritance GuessGame.java

```
package cc.openhome;

public abstract class GuessGame {
    public void go() {
        int number = (int) (Math.random() * 10);
        int guess;
        do {
            print("输入数字：");
            guess = nextInt();
        } while(guess != number);
        println("猜中了");
    }

    public void println(String text) {
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        print(text + "\n");
    }

        public abstract void print(String text);
        public abstract int nextInt();
}
```

这个类的定义不完整，print()、println()与nextInt()都是抽象方法，因为老板还没决定在哪个环境执行猜数字游戏，所以如何显示输出、取得用户输入就不能操作。可以先操作的是猜数字的流程，虽然是抽象方法，但在go()方法中，还是可以调用。

等到第二星期开会决定，还是在文本模式下执行猜数字游戏，你再撰写ConsoleGame类，继承抽象类GuessGame，操作当中的抽象方法即可。

Inheritance_ConsoleGame.java

```
package cc.openhome;

import java.util.Scanner;

public class ConsoleGame extends GuessGame {
    private Scanner scanner = new Scanner(System.in);

    @Override
    public void print(String text) {
        System.out.print(text);
    }

    @Override
    public void println(String text) {
        System.out.println(text);
    }

    @Override
    public int nextInt() {
        return scanner.nextInt();
    }
}
```

实际上只要创建出ConsoleGame实例，执行go()方法过程中调用到print()、nextInt()或println()等方法时，都是执行ConsoleGame中定义的流程，完整的猜数字游戏就操作出来了。例如：

Inheritance_Guess.java

```
package cc.openhome;

public class Guess {
    public static void main(String[] args) {
        GuessGame game = new ConsoleGame();
        game.go();
    }
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

其执行的结果如下：

```
输入数字：5
输入数字：4
输入数字：3
猜中了
```

提示 »» 设计上的经验，称为设计模式(Design Pattern)，上面的例子是 Template Method 模式的实例。如果对其他设计模式有兴趣，可以先从文章《设计模式》开始：

http://openhome.cc/Gossip/DesignPattern/DesignPattern.htm

6.3 重点复习

面向对象中，子类继承父类，避免重复的行为定义，不过并非为了避免重复定义行为就使用继承。如何正确判断使用继承的时机，以及继承之后如何活用多态，才是学习继承时的重点。

程序代码重复在程序设计上就是不好的信号，多个类间出现重复的程序代码时，设计上可考虑的改进方式之一，就是把相同的程序代码提升为父类。

在 Java 中，继承时使用 `extends` 关键字，`private` 成员也会被继承，只不过子类无法直接存取，必须通过父类提供的方法来存取(如果父类愿意提供访问方法的话)。

在 Java 中，子类只能继承一个父类，继承有个重要的关系，就是子类与父类间会有 is-a 的关系。要开始理解多态，必须先知道你操作的对象是“哪一种”东西。

检查多态语法逻辑是否正确，方式是从=号右边往左读：右边是不是一种左边呢(右边类型是不是左边类型的子类)？如果不是就会编译失败，如果加上扮演(Cast)语法，编译程序就让程序代码通过编译，不过后果得自行负责，也就是扮演失败，执行时会抛出 `ClassCastException`。

什么叫多态？以抽象讲法解释，就是使用单一接口操作多种类型的对象。若用 6.1.2 节的范例来理解，在 `showBlood()`方法中，既可以通过 `Role` 类型操作 `SwordsMan` 对象，也可以通过 `Role` 类型操作 `Magician` 对象。

如果某方法区块中真的没有任何程序代码操作，可以使用 `abstract` 标示该方法为抽象方法，该方法不用撰写`{}`区块，直接“`;`”结束即可。类中若有方法没有操作，并且标示为 `abstract`，表示这个类定义不完整，定义不完整的类就不能用来生成实例。Java 中规定内含抽象方法的类，一定要在 `class` 前标示 `abstract`，表示这是一个定义不完整的抽象类。

被声明为 `protected` 的成员，相同包中的类可以直接存取，不同包中的类可以在继承后的子类直接存取。

Java 中有 `public`、`protected` 与 `private` 三个权限关键字，但实际上有四个权限范围。

如果想取得父类中的方法定义，可以在调用方法前，加上 `super` 关键字。重新定义方法要注意，对于父类中的方法权限，只能扩大不能缩小。在 JDK5 之后，重新定义方法时，如果返回类型是父类中方法返回类型的子类，也是可以通过编译的。

如果子类构造函数中没有指定执行父类中哪个构造函数，默认会调用父类中无参数构造函数。如果想执行父类中某构造函数，可以使用 `super()`指定。`this()`与 `super()`只能择一调用，而且一定要在构造函数第一行执行。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

如果 class 前使用了 final 关键字定义，那么表示这个类是最后一个了，不会再有子类，也就是不能被继承。定义方法时，也可以限定该方法为 final，这表示最后一次定义方法，也就是子类不可以重新定义 final 方法。

如果定义类时没有使用 extends 关键字指定继承任何类，那一定是继承 java.lang.Object。在 Java 中，任何类追溯至最上层父类，一定就是 java.lang.Object。

对于不再有用的对象，JVM 有垃圾收集机制，收集到的垃圾对象所占据的内存空间，会被垃圾收集器释放。执行流程中，无法通过变量参考的对象，就是 GC 认定的垃圾对象。

6.4 课后练习

1. 如果使用 6.2.5 节设计的 ArrayList 类收集对象，想显示所收集对象的字符串描述时，必须如下：

```
  ArrayList list = new ArrayList();
//...收集对象
for(int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
  }
```

请重新定义 ArrayList 的 toString()方法，让客户端想显示所收集对象的字符串描述时，可以如下：

```
ArrayList list = new ArrayList();
//...收集对象
  System.out.println(list);
```

2. 承上题，若想比较两个 ArrayList 实例是否相等，希望可以如下比较：

```
  ArrayList list1 = new ArrayList();
//...用 list1 收集对象
ArrayList list2 = new ArrayList();
//...用 list2 收集对象
  System.out.println(list1.equals(list2));
```

请重新定义 ArrayList 的 equals()方法，先比较收集的对象个数，再比较各索引的对象实质上是否相等(使用各对象的 equals()比较)。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 7

接口与多态

学习目标

- 使用接口定义行为
- 了解接口的多态操作
- 利用接口枚举常数
- 利用 enum 枚举常数

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

7.1 何谓接口

第 6 章介绍过继承，但你也许在一些书或文件中看过“别滥用继承”，或者“优先考虑接口而不是用继承”的说法。什么情况叫滥用继承？接口代表的又是什么？这一节我们通过一个海洋乐园游戏，看看如何设计与改进，介绍继承、接口与多态的用与不用之处。

7.1.1 接口定义行为

老板今天想开发一个海洋乐园游戏，当中所有东西都会游泳。你想了一下，谈到会游的东西，第一个想到的就是鱼，上一章刚学过继承，也知道继承可以运用多态，你也许会定义 Fish 类中有个 swim() 的行为：

```
public abstract class Fish {
    protected String name;
    public Fish(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public abstract void swim();
}
```

由于实际上每种鱼游泳方式不同，所以将 swim() 定义为 abstract，因此 Fish 也是 abstract。接着定义小丑鱼继承鱼：

```
public class Anemonefish extends Fish {
   public Anemonefish(String name) {
       super(name);
   }
    @Override
    public void swim() {
        System.out.printf("小丑鱼 %s 游泳%n", name);
    }
}
```

Anemonefish 继承了 Fish，并操作 swim() 方法，也许你还定义了鲨鱼 Shark 类继承 Fish、食人鱼 Piranha 继承 Fish：

```
public class Shark extends Fish {
   public Shark(String name) {
       super(name);
   }
    @Override
    public void swim() {
        System.out.printf("鲨鱼 %s 游泳%n", name);
    }
}


public class Piranha extends Fish {
   public Piranha(String name) {
       super(name);
```

```
    }
    @Override
    public void swim() {
        System.out.printf("食人鱼 %s 游泳%n", name);
    }
}
```

老板说话了，为什么都是鱼？人也会游泳啊！怎么没写？于是你再定义 Human 类继承 Fish。等一下，Human 继承 Fish？不会觉得很奇怪吗？你会说程序没错啊！编译程序也没抱怨什么。

对！编译程序是不会抱怨什么，就目前为止，程序也可以执行，但是请回想上一章的内容，继承会有“是一种”(is-a)的关系，所以 Anemonefish 是一种 Fish，Shark 是一种 Fish，Piranha 是一种 Fish，如果你让 Human 继承 Fish，那 Human 是一种 Fish？你会说“美人鱼啊!”，如图 7.1 所示。

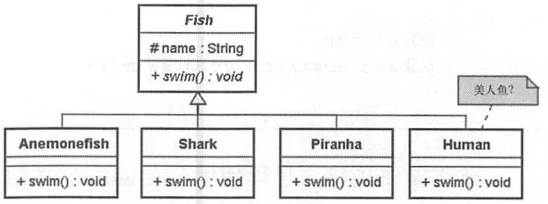


图 7.1　继承会有 is-a 关系

提示 »» 图 7.1 中 Fish 是斜体字，表示是抽象类，而 swim() 是斜体字，表示是抽象方法。name 旁边有个#，表示为 protected。

程序上可以通过编译也可以执行，但逻辑上或设计上有不合理的地方，或许你可以继续硬掰下去，但是如果现在老板说加个潜水艇呢？写个 Submarine 继承 Fish 吗？Submarine 是一种 Fish 吗？继续按照这样的想法设计下去，你的程序架构会越来越不合理，越来越没有弹性。

记得吗？Java 中只能继承一个父类，所以更强化了“是一种”关系的限制性。如果今天老板突发奇想，想把海洋乐园变为海空乐园，有的东西会游泳，有的东西会飞，有的东西会游也会飞，如果用继承方式来解决，写个 Fish 让会游的东西继承，写个 Bird 让会飞的东西继承，那会游也会飞的怎么办？有办法定义个飞鱼 FlyingFish 同时继承 Fish 与 Bird 吗？

重新想一下需求吧！老板今天想开发一个海洋乐园游戏，当中所有东西都会游泳。“所有东西”都会“游泳”，而不是“某种东西”都会“游泳”，前面的设计方式只解决了“所有鱼”都会“游泳”，只要它是一种鱼(也就是继承 Fish)。

“所有东西”都会“游泳”，代表了“游泳”这个“行为”可以被所有东西拥有，而不是“某种”东西专属。对于“定义行为”，在 Java 中可以使用 **interface** 关键字定义：

OceanWorld1 Swimmer.java

```
package cc.openhome;

public interface Swimmer {
    public abstract void swim();
}
```

以下程序代码定义了 Swimmer 接口，接口可以用于定义行为但不定义操作，在这里 Swimmer 中的 swim() 方法没有操作，直接标示为 abstract，而且一定是 public。对象若想拥有 Swimmer 定义的行为，就必须操作 Swimmer 接口。例如，Fish 拥有 Swimmer 行为：

OceanWorld1 Fish.java

```
package cc.openhome;

public abstract class Fish implements Swimmer {
    protected String name;
    public Fish(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    @Override
    public abstract void swim();
}
```

类要操作接口，必须使用 **implements** 关键字。操作某接口时，对接口中定义的方法有两种处理方式，一是操作接口中定义的方法，二是再度将该方法标示为 abstract。在这个范例中，由于 Fish 并不知道每条鱼怎么游，所以使用第二种处理方式。

目前 Anemonefish、Shark 与 Piranha 继承 Fish 后的程序代码如同前面示范的片段，无须修改。那么，如果 Human 要能游泳呢？

OceanWorld1 Human.java

```
package cc.openhome;

public class Human implements Swimmer {
    private String name;
    public Human(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    @Override
    public void swim() {
        System.out.printf("人类 %s 游泳%n", name);
    }
}
```

Human 操作了 Swimmer，不过这次 Human 没有继承 Fish，所以 Human 不是一种 Fish。类似地，Submarine 也有 Swimmer 的行为：

OceanWorld1 Submarine.java

```
package cc.openhome;

public class Submarine implements Swimmer {
    private String name;
    public Submarine(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
```

```
    @Override
    public void swim() {
        System.out.printf("潜水艇 %s 潜行%n", name);
    }
}
```

Submarine 操作了 Swimmer，不过 Submarine 没有继承 Fish，所以 Submarine 不是一种 Fish。目前程序的架构如图 7.2 所示。

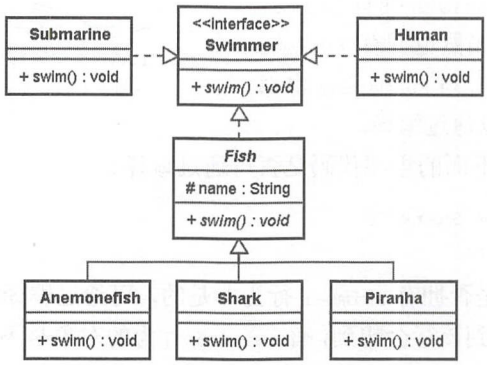


图 7.2 运用接口改进程序架构

提示 »» 图 7.2 中的虚线空心箭头表示运用接口。

以 Java 的语意来说，继承会有“是一种”关系，操作接口则表示“拥有行为”，但不会有“是一种”的关系。Human 与 Submarine 操作了 Swimmer，所以都拥有 Swimmer 定义的行为，但它们没有继承 Fish，所以它们不是一种鱼，这样的架构比较合理也较有弹性，可以应付一定程度的需求变化。

提示 »» 有些书或文件认为，Human 与 Submarine 是一种 Swimmer。会有这种想法的作者，应该是有 C++程序语言的背景，因为 C++中可以多重继承，也就是子类可以拥有两个以上的父类，若其中一个父类用来定义抽象行为，该父类的作用就类似 Java 中的接口，因为也是用继承语意来操作，所以才会有“是一种”的说法。

多重继承容易因为设计上考虑不周而引来不少麻烦，因而 Java 对多重继承做了限制，就类别的语意来说，Java 中限制只能继承一个父类别，所以“是一种”的语意更为强烈。建议将“是一种”的语意保留给继承，对于接口操作则使用“拥有行为”的语意，这样就不会搞不清楚类继承与接口操作的差别，对于何时用继承、何时用接口也比较容易判断。

广义来说，Java 的接口确实是支持多重继承的一种方式，不过在 JDK8 出现前，Java 的接口只能定义抽象方法，不能有任何方法实现，这也是 Java 对多重继承做限制以避免复杂度的表现，但也引来设计上的一些不便之处。为了支持 Lambda 新特性的引入，从 JDK8 开始，Java 的接口也放宽了一些限制，接口中也可以有条件地进行方法实现，这在第 12 章介绍 Lambda 时会提到。

7.1.2 行为的多态

在 6.1.2 节曾试着通过编译程序判断哪些继承多态语法可以通过编译，了解加入扮演(Cast)语法的目的是为何，以及哪些情况下，执行时期会扮演失败，哪些又可扮演成功。学会使用接口定义行为之后，也要再来利用编译程序，看看哪些是合法的多态语法。例如：

```
Swimmer swimmer1 = new Shark();
Swimmer swimmer2 = new Human();
Swimmer swimmer3 = new Submarine();
```

这三行程序代码都可以通过编译，判断方式是“右边是不是拥有左边的行为”，或者“右边对象是不是操作了左边接口”，如图 7.3 所示。

图 7.3 是否拥有行为？是否操作接口？

Shark 拥有 Swimmer 行为吗？有。因为 Fish 操作了 Swimmer 接口，也就是 Fish 拥有 Swimmer 行为，Shark 继承 Fish，当然也拥有 Swimmer 行为，所以通过编译，Human 与 Submarine 也都操作了 Swimmer 接口，所以通过编译。

更进一步地，来看看下面的程序代码是否可通过编译？

```
Swimmer swimmer = new Shark();
Shark shark = swimmer;
```

第一行要判断 Shark 是否拥有 Swimmer 行为？是的，可通过编译。但第二行呢？swimmer 是 Swimmer 类型，编译程序看到该行会想到，有 Swimmer 行为的对象是不是 Shark 呢？这可不一定，也许实际上是 Human 实例。因为有 Swimmer 行为的对象不一定是 Shark，所以第二行编译失败。

就上面的代码段而言，实际上 swimmer 是参考至 Shark 实例。可以加上扮演(Cast)语法：

```
Swimmer swimmer = new Shark();
Shark shark = (Shark) swimmer;
```

对第二行的语意而言，就是在告诉编译程序，对！你知道有 Swimmer 行为的对象，不一定是 Shark，不过你就是要它扮演 Shark，所以编译程序就别再啰唆了。可以通过编译，执行时期 swimmer 确实也是参考 Shark 实例，所以也没有错误。

下面的程序片段会在第二行编译失败：

```
Swimmer swimmer = new Shark();
Fish fish = swimmer;
```

第二行 swimmer 是 Swimmer 类型，所以编译程序会问，操作 Swimmer 接口的对象是不是继承 Fish？不一定，也许是 Submarine。因为会操作 Swimmer 接口的并不一定继承 Fish，所以编译失败了。如果加上扮演语法：

```
Swimmer swimmer = new Shark();
Fish fish = (Fish) swimmer;
```

第二行告诉编译程序，你知道有 Swimmer 行为的对象，不一定继承 Fish，不过你就是要它扮演 Fish，编译程序就别再啰唆了。可以通过编译，执行时期 swimmer 确实也是参考 Shark 实例，它是一种 Fish，所以也没有错误。

下面这个例子就会抛出 **ClassCastException** 错误：

```
Swimmer swimmer = new Human();
Shark shark = (Shark) swimmer;
```

在第二行，swimmer 实际上参考了 Human 实例，你要他扮演鲨鱼？这太荒谬了，所以执行时就出错了。类似地，下面的例子也会出错：

```
Swimmer swimmer = new Submarine();
Fish fish = (Fish) swimmer;
```

在第二行，swimmer 实际上参考了 Submarine 实例，你要它扮演鱼？又不是在演哆啦 A 梦中的海底鬼岩城，哪来的机器鱼情节。Submarine 不是一种 Fish，执行时期会因为扮演失败而抛出 ClassCastException 错误。

知道以下的语法，哪些可以通过编译，哪些可以扮演成功做什么。来考虑一个需求，写个 static 的 doSwim() 方法，让会游的东西都游起来，在不会使用接口多态语法时，也许你会写下：

```
public static void doSwim(Fish fish) {
    fish.swim();
}
public static void doSwim(Human human) {
    human.swim();
}
public static void doSwim(Submarine submarine) {
    submarine.swim();
}
```

老实说，如果已经会写接收 Fish 的 doSwim() 版本，程序还算是不错的，因为至少你知道，只要继承 Fish，无论 Anemonefish、Shark 还是 Piranha，都可以使用 Fish 的 doSwim() 版本，至少你会使用继承时的多态。至于 Human 与 Submarine 各使用其接收 Human 及 Submarine 的 doSwim() 版本。

Lab

问题是，如果“种类”很多怎么办？多了水母、海蛇、虫等种类呢？每个种类重载一个方法出来吗？其实在你的设计中，会游泳的东西都拥有 Swimmer 的行为，都操作了 Swimmer 接口，所以你只要这么设计就可以了：

OceanWorld2 Ocean.java

```
package cc.openhome;

public class Ocean {
    public static void main(String[] args) {
        doSwim(new Anemonefish("尼莫"));
        doSwim(new Shark("兰尼"));
        doSwim(new Human("贾斯汀"));
        doSwim(new Submarine("黄色一号"));
    }

    static void doSwim(Swimmer swimmer) { ←❶ 参数是 Swimmer 型态
        swimmer.swim();
    }
}
```

执行结果如下：

```
小丑鱼 尼莫 游泳
鲨鱼 兰尼 游泳
人类 贾斯汀 游泳
潜水艇 黄色一号 潜行
```

只要是操作 Swimmer 接口的对象，都可以使用范例中的 doSwim() 方法，Anemonefish、Shark、Human、Submarine 等都操作了 Swimmer 接口。再多种类，只要对象拥有 Swimmer 行为，便不用撰写新的方法，可维护性显然提高许多。

7.1.3 解决需求变化

常听人家说，写程序要有弹性，要有可维护性。那什么叫有弹性？何谓可维护？老实说，这是有点抽象的问题。这里从最简单的定义开始：如果增加新的需求，原有的程序无须修改，只需针对新需求撰写程序，那就是有弹性、具可维护性的程序。

以 7.1.1 节提到的需求为例，如果今天老板突发奇想，想把海洋乐园变为海空乐园，有的东西会游泳，有的东西会飞，有的东西会游也会飞，那么现有的程序可以应付这个需求吗？

仔细想想，有的东西会飞，但不限于某种东西才有“飞”这个行为。有了前面的经验，你使用 interface 定义了 Flyer 接口：

OceanWorld3 Flyer.java

```
package cc.openhome;

public interface Flyer {
    public abstract void fly();
}
```

Flyer 接口定义了 fly()方法，程序中想要飞的东西，可以操作 Flyer 接口。假设有架海上飞机具有飞行的行为，也可以在海面上航行，可以定义 Seaplane 操作 Swimmer 与 Flyer 接口：

OceanWorld3 Seaplane.java

```
package cc.openhome;

public class Seaplane implements Swimmer, Flyer {
    private String name;

    public Seaplane(String name) {
        this.name = name;
    }

    @Override
    public void fly() {
        System.out.printf("海上飞机 %s 在飞%n", name);
    }

    @Override
    public void swim() {
        System.out.printf("海上飞机 %s 航行海面%n", name);
    }
}
```

在 Java 中，类可以操作两个以上的类，也就是拥有两种以上的行为。例如，Seaplane 同时拥有 Swimmer 与 Flyer 的行为。

如果是会游也会飞的飞鱼呢？飞鱼是一种鱼，可以继承 Fish 类，飞鱼会飞，可以操作 Flyer 接口：

OceanWorld3 FlyingFish.java

```
package cc.openhome;

public class FlyingFish extends Fish implements Flyer {
    public FlyingFish(String name) {
```

```
        super(name);
    }

    @Override
    public void swim() {
        System.out.println("飞鱼游泳");
    }

    @Override
    public void fly() {
        System.out.println("飞鱼会飞");
    }
}
```

正如范例所示，在 Java 中，类可以同时继承某个类，并操作某些接口。例如，FlyingFish 是一种鱼，也拥有 Flyer 的行为。如果现在要让所有会游的东西游泳，那么 7.1.1 节中的 doSwim() 方法就可以满足需求了，因为 Seaplane 拥有 Swimmer 的行为，而 FlyingFish 也拥有 Swimmer 的行为：

OceanWorld3 Ocean.java

```
package cc.openhome;
public class Ocean {
    public static void main(String[] args) {
        ...
        doSwim(new Seaplane("空军零号"));
        doSwim(new FlyingFish("甚平"));
    }

    static void doSwim(Swimmer swimmer) {
        swimmer.swim();
    }
}
```

就满足目前需求来说，你所做的就是新增程序代码来满足需求，但没有修改旧有既存的程序代码，你的程序确实拥有某种程度的弹性与可维护性，如图 7.4 所示。

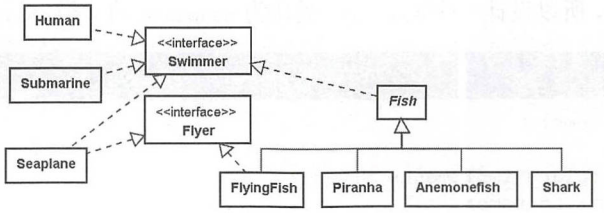


图 7.4　海空乐园目前设计架构

当然需求是无止境的，原有程序架构也许确实可满足某些需求，但有些需求也可能超过了原有架构预留的弹性，一开始要如何设计才会有弹性，得靠经验与分析判断，不用为了保有程序弹性的弹性而过度设计。因为过大的弹性表示过度预测需求，有的设计也许从不会遇上事先假设的需求。

例如，也许你预先假设会遇上某些需求而设计了一个接口，但从程序开发至生命周期结束，该接口从未被操作过，或者仅有一个类操作过该接口，那么该接口也许就不必存在，你事先的假设就是过度预测需求。

事先的设计也有可能因为需求不断增加，而超出原本预留的弹性。例如，老板又开口了：不是所有的人都会游泳啊！有的飞机只会飞，不能停在海上啊！

好吧！并非所有的人都会游泳，所以不再让 Human 操作 Swimmer：

OceanWorld4 Human.java

```
package cc.openhome;

public class Human {
    protected String name;

    public Human(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

假设只有游泳选手会游泳，游泳选手是一种人，并拥有 Swimmer 的行为：

OceanWorld4 SwimPlayer.java

```
package cc.openhome;

public class SwimPlayer extends Human implements Swimmer {
    public SwimPlayer(String name) {
        super(name);
    }

    @Override
    public void swim() {
        System.out.printf("游泳选手 %s 游泳%n", name);
    }
}
```

有的飞机只会飞，所以设计一个 Airplane 类作为 Seaplane 的父类，Airplane 操作 Flyer 接口：

OceanWorld4 Airplane.java

```
package cc.openhome;

public class Airplane implements Flyer {
    protected String name;

    public Airplane(String name) {
        this.name = name;
    }

    @Override
    public void fly() {
        System.out.printf("飞机 %s 在飞%n", name);
    }
}
```

Lab

Seaplane 会在海上航行，所以在继承 Airplane 之后，必须操作 Swimmer 接口：

OceanWorld4 Seaplane.java

```
package cc.openhome;

public class Seaplane extends Airplane implements Swimmer {
    public Seaplane(String name) {
        super(name);
    }

    @Override
    public void fly() {
        System.out.print("海上");
        super.fly();
    }

    @Override
    public void swim() {
        System.out.printf("海上飞机 %s 航行海面%n", name);
    }
}
```

Lab

不过程序中的直升机就只会飞：

OceanWorld4 Seaplane.java

```
package cc.openhome;

public class Helicopter extends Airplane {
    public Helicopter(String name) {
        super(name);
    }

    @Override
    public void fly() {
        System.out.printf("飞机 %s 在飞%n", name);
    }
}
```

这一连串的修改都是为了调整程序架构，这只是个简单的示范。想象一下，在更大规模的程序中调整程序架构多么麻烦，而且不只是修改程序很麻烦，没有被修改到的地方也有可能因此出错，如图 7.5 所示。

```
public stati
    doSwim(
    doSwim(
    doSwim(new Human("贾斯汀"));
    doSwim(new Submarine("黄色一号"));
    doSwim(new Seaplane("空军零号"));
    doSwim(new FlyingFish("甚平"));
```

incompatible types: Human cannot be converted to Swimmer

(Alt-Enter shows hints)

图 7.5 没有动到这里啊！怎么出错了？

程序架构很重要。这里就是个例子，因为 Human 不再操作 Swimmer 接口了，因此不能再套用 doSwim() 方法，应该改用 SwimPlayer。

非卖品，仅供非商业用途或交流学习使用

不好的架构下要修改程序，很容易牵一发而动全身。想象一下，在更复杂的程序中修改程序之后到处出错的窘境，也有不少人维护到架构不好的程序，抓狂到想重写的情况。

提示 »» 对于一些人来说，软件看不到、摸不着，改程序似乎也不需成本，因此架构这东西经常被漠视。曾经听过一个比喻是这样的：没人敢在盖十几层高楼之后，要求修改地下室架构，但软件业界常常在做这种事。

也许老板又想到了：水里的话，将浅海游泳与深海潜行分开好了。就算心里有千百个不愿意，你还是改了：

OceanWorld4 Diver.java

```
package cc.openhome;

public interface Diver extends Swimmer {
    public abstract void dive();
}
```

在 Java 中，接口可以继承自另一个接口，也就是继承父接口行为，再在子接口中额外定义行为。假设一般的船可以在浅海航行：

OceanWorld4 Boat.java

```
package cc.openhome;

public class Boat implements Swimmer {
    protected String name;

    public Boat(String name) {
        this.name = name;
    }

    @Override
    public void swim() {
        System.out.printf("船在水面 %s 航行%n", name);
    }
}
```

潜水艇是一种船，可以在浅海游泳，也可以在深海潜行：

OceanWorld4 Submarine.java

```
package cc.openhome;

public class Submarine extends Boat implements Diver {
    public Submarine(String name) {
        super(name);
    }

    @Override
    public void dive() {
        System.out.printf("潜水艇 %s 潜行%n", name);
    }
}
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

需求不断变化，架构也有可能因此而修改。好的架构在修改时，并不会全部的程序代码都被牵动，这就是设计的重要性。不过像这位老板无止境地在扩张需求(见图 7.6)，他说一个你改一个，也不是办法，找个时间，好好跟老板谈谈这个程序的需求边界到底在哪儿吧。

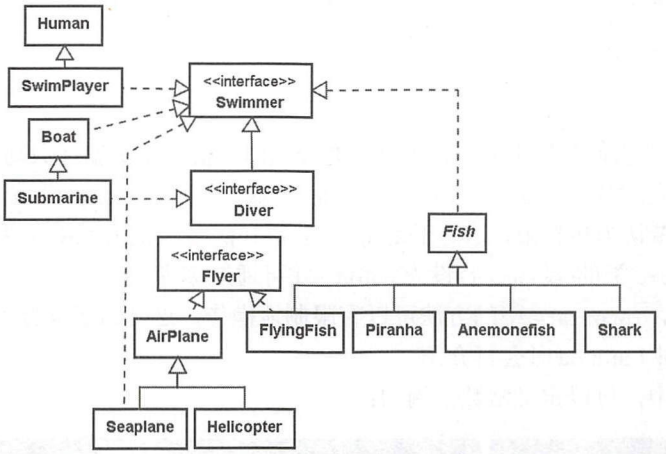


图 7.6　你的程序有弹性吗？

7.2　接口语法细节

上一节介绍了接口的基础概念与语法，然而结合 Java 的特性，接口还有一些细节必须明了，像是接口中的默认语法、接口中定义常数的运用、匿名内部类的撰写等，这都将在本节中详细说明。

7.2.1　接口的默认

在 Java 中，可使用 `interface` 来定义抽象的行为与外观，如接口中的方法可声明为 **`public abstract`**。例如：

```
public interface Swimmer {
    public abstract void swim();
}
```

接口中的方法在没有操作时，一定得是公开且抽象，为了方便，也可以省略 **`public abstract`**：

```
public interface Swimmer {
    void swim();  // 默认就是 public abstract
}
```

编译程序会自动加上 `public abstract`。由于默认一定是 `public`，因此认证考试上经常会出这个题目：

```
interface Action {
    void execute();
}

class Some implements Action {
    void execute() {
```


严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

```
            System.out.println("做一些服务");
        }
    }
    public class Main {
        public static void main(String[] args) {
            Action action = new Some();
            action.execute();
        }
    }
```

“请问执行结果为何？”这个问题本身就是个陷阱，根本无法编译成功，因为 Action 中定义的 execute()其实默认为 public abstract，而 Some 类在操作 execute()方法时，没有撰写 public，因此就是默认为包权限，这等于是将 Action 中 public 的方法缩小为包权限，所以编译失败了。必须将 Some 类的 execute()设为 public 才可通过编译。

从 JDK8 开始，interface 中的方法可以有限制地操作，这是为了支持 Lambda 而扩充的新特性，第 12 章在讲 Lambda 时会再介绍。

在 interface 中，可以定义常数。例如：

Interface Action.java

```
package cc.openhome;

public interface Action {
    public static final int STOP = 0;
    public static final int RIGHT = 1;
    public static final int LEFT = 2;
    public static final int UP = 3;
    public static final int DOWN = 4;
}
```

Java 中经常见到在接口中定义这类常数，称为枚举常数，这使程序看起来更清晰一些。例如：

Interface Game.java

```
package cc.openhome;

import static java.lang.System.out;

public class Game {
    public static void main(String[] args) {
        play(Action.RIGHT);
        play(Action.UP);
    }

    public static void play(int action) {
        switch(action) {
            case Action.STOP:
                out.println("播放停止动画");
                break;
            case Action.RIGHT:
                out.println("播放向右动画");
                break;
            case Action.LEFT:
                out.println("播放向左动画");
                break;
            case Action.UP:
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
                out.println("播放向上动画");
                break;
            case Action.DOWN:
                out.println("播放向下动画");
                break;
            default:
                out.println("不支持此动作");
        }
    }
}
```

如果将上面这个程序改为以下写法，哪个在维护程序时更方便呢？

```
...
    public static void play(int action) {
        switch(action) {
            case 0:    // 数字比较清楚？还是枚举常数比较清楚？
                out.println("播放停止动画");
                break;
            case 1:
                out.println("播放向右动画");
                break;
            case 2:
                out.println("播放向左动画");
                break;
            ...
        }
    }
    public static void main(String[] args) {
        play(1);      // 数字比较清楚？还是枚举常数比较清楚？
        play(3);
    }
...
```

事实上，在 interface 中，也只能定义 public static final 的枚举常数。为了方便，也可以如下撰写：

```
public interface Action {
    int STOP = 0;
    int RIGHT = 1;
    int LEFT = 2;
    int UP = 3;
    int DOWN = 4;
}
```

编译程序会展开为 public static final，所以在接口中枚举常数，一定要使用=指定值，否则就会编译错误，如图 7.7 所示。

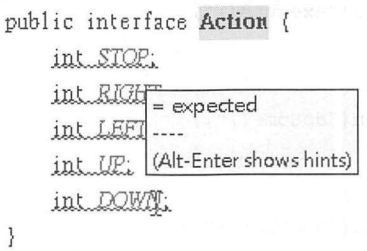


图 7.7 接口枚举常数一定是 public static final

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

提示>>> 要在类中定义枚举常数也是可以的，不过一定要明确写出 public static final。

类可以操作两个以上的接口，如果有两个接口都定义了某方法，而操作两个接口的类会怎样呢？程序面上来说，并不会有错误，照样通过编译：

```
interface Some {
    void execute();
    void doSome();
}
interface Other {
    void execute();
    void doOther();
}
public class Service implements Some, Other {
    @Override
    public void execute() {
        System.out.println("execute()");
    }
    @Override
    public void doSome() {
        System.out.println("doSome()");
    }
    @Override
    public void doOther() {
        System.out.println("doOther()");
    }
}
```

但在设计上，要思考一下：**Some** 与 **Other** 定义的 **execute()** 是否表示不同的行为？

如果表示不同的行为，那么 Service 在操作时，应该有不同的方法操作，Some 与 Other 的 execute() 方法就得在名称上有所不同，Service 在操作时才可以有两个不同的方法操作。

如果表示相同的行为，可以定义一个父接口，再在当中定义 execute() 方法，而 Some 与 Other 继承该接口，各自定义自己的 doSome() 与 doOther() 方法：

```
interface Action {
    void execute();
}
interface Some extends Action {
    void doSome();
}
interface Other extends Action {
    void doOther();
}
public class Service implements Some, Other {
    @Override
    public void execute() {
        System.out.println("execute()");
    }
    @Override
    public void doSome() {
        System.out.println("doSome()");
    }
    @Override
    public void doOther() {
        System.out.println("doOther()");
    }
}
```

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

接口可以继承别的接口，也可以同时继承两个以上的接口，同样也是使用 extends 关键字，这代表了继承父接口的行为。在 JDK8 之后，如果父接口中定义的方法有操作，也代表了继承父接口的操作。

7.2.2 匿名内部类

在撰写 Java 程序时，经常会有临时继承某个类或操作某个接口并建立实例的需求。由于这类子类或接口操作类只使用一次，不需要为这些类定义名称，这时可以使用匿名内部类(Anonymous Inner Class)来解决这个需求。匿名内部类的语法为：

```
new 父类()|接口() {
    // 类本体操作
};
```

在 5.2.7 节的内部类中简单介绍过匿名内部类，那时以继承 Object 重新定义 toString()方法为例：

```
Object o = new Object() {  // 继承 Object 重新定义 toString()并直接产生实例
    @Override
    public String toString() {
        return "无聊的语法示范而已";
    }
};
```

如果是操作某个接口，例如 Some 接口定义了 doService()方法，要建立匿名类实例，可以如下撰写：

```
Some some = new Some() {  // 操作 Some 接口并直接产生实例
    public void doService() {
        System.out.println("做一些事");
    }
};
```

提示 »» 实际上，从 JDK8 开始，若接口仅定义一个抽象方法，可以使用 Lambda 表示式来简化这个程序的撰写。例如：`Some some = () -> { out.println("做一些事");};`
有关 Lambda 语法的细节，第 9 章与第 12 章还会说明。

举个接口应用的例子。假设要开发多人联机程序，对每个联机客户端，都会建立 Client 对象封装相关信息：

Interface Client.java

```
package cc.openhome;

public class Client {
    public final String ip;
    public final String name;
    public Client(String ip, String name) {
        this.ip = ip;
        this.name = name;
    }
}
```

程序中建立的 Client 对象，都会加入 ClientQueue 集中管理，若程序中其他部分希望在 ClientQueue 的 Client 加入或移除时可以收到通知，以便做一些处理(例如进行日志记录)，那么

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

可以将 Client 加入或移除的信息包装为 ClientEvent：

Interface ClientEvent.java

```
package cc.openhome;

public class ClientEvent {
    private Client client;
    public ClientEvent(Client client) {
        this.client = client;
    }

    public String getName() {
        return client.name;
    }

    public String getIp() {
        return client.ip;
    }
}
```

可以定义 ClientListener 接口，如果有对象对 Client 加入 ClientQueue 有兴趣，可以操作这个接口：

Interface ClientListener.java

```
package cc.openhome;

public interface ClientListener {
    void clientAdded(ClientEvent event);     // 新增 Client 会调用这个方法
    void clientRemoved(ClientEvent event);   // 移除 Client 会调用这个方法
}
```

如何在 ClientQueue 新增或移除 Client 时予以通知呢？来看程序代码：

Interface ClientQueue.java

```
package cc.openhome;

import java.util.ArrayList;

public class ClientQueue {
    private ArrayList clients = new ArrayList();      <-- ❶ 收集联机的 Client
    private ArrayList listeners = new ArrayList();    <-- ❷ 收集对 ClientQueue 有兴趣的 ClientListener

    public void addClientListener(ClientListener listener) {   <-- ❸ 注册
        listeners.add(listener);
    }

    public void add(Client client) {
        clients.add(client);                          <-- ❹ 新增 Client
        ClientEvent event = new ClientEvent(client);  <-- ❺ 通知信息包装为 ClientEvent
        for(int i = 0; i < listeners.size(); i++) {
            ClientListener listener = (ClientListener) listeners.get(i);
            listener.clientAdded(event);              <-- ❻ 逐一取出 ClientListener 通知
        }
    }
```


严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

```
    public void remove(Client client) {
        clients.remove(client);
        ClientEvent event = new ClientEvent(client);
        for(int i = 0; i < listeners.size(); i++) {
            ClientListener listener = (ClientListener) listeners.get(i);
            listener.clientRemoved(event);
        }
    }
}
```

ClientQueue 会收集联机后的 Client 对象，在 6.2.5 节曾经自己定义过 ArrayList 类。在第 9 章会介绍到，Java SE API 提供了 java.util.ArrayList，可以让你进行对象收集，范例中就使用了 java.util.ArrayList 来收集 Client❶，以及对 ClientQueue 感兴趣的 ClientListener❷。

如果有对象对 Client 加入 ClientQueue 有兴趣，可以操作 ClientListener，并通过 addClientListener()注册❸。当每个 Client 通过 ClientQueue 的 add()收集时，会用 ArrayList 收集 Client❹，接着使用 ClientEvent 封装 Client 相关信息❺，再使用 for 循环将注册的 ClientListener 逐一取出，并调用 clientAdded()方法进行通知❻。如果有对象被移除，流程也是类似，则可以在 ClientQueue 的 remove()方法中看到相关程序代码。

作为测试，可以使用以下的程序代码，其中使用匿名内部类，直接建立操作 ClientListener 的对象：

Interface MultiChat.java

```
package cc.openhome;

public class MultiChat {
    public static void main(String[] args) {
        Client c1 = new Client("127.0.0.1", "Caterpillar");
        Client c2 = new Client("192.168.0.2", "Justin");

        ClientQueue queue = new ClientQueue();
        queue.addClientListener(new ClientListener() {
            @Override
            public void clientAdded(ClientEvent event) {
                System.out.printf("%s 从 %s 联机%n",
                        event.getName(), event.getIp());
            }

            @Override
            public void clientRemoved(ClientEvent event) {
                System.out.printf("%s 从 %s 脱机%n",
                        event.getName(), event.getIp());
            }
        });

        queue.add(c1);
        queue.add(c2);
        queue.remove(c1);
        queue.remove(c2);
    }
}
```


严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

执行的结果如下所示：

```
caterpillar 从 127.0.0.1 联机
justin 从 192.168.0.2 联机
caterpillar 从 127.0.0.1 脱机
justin 从 192.168.0.2 脱机
```

在 JDK8 出现前，如果要在匿名内部类中存取局部变量，则该局部变量必须是 **final**，否则会发生编译错误，如图 7.8 所示。

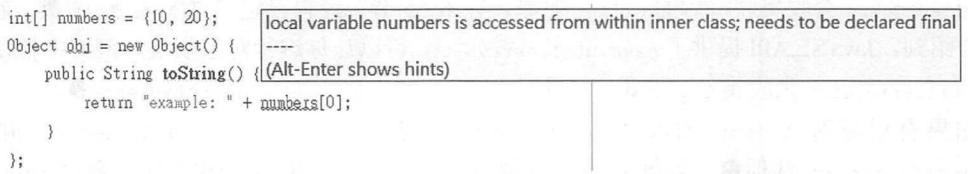


图 7.8　在 JDK8 出现前，内部类只能取得 final 局部变量

必须声明 numbers 为 final 才可以通过编译：

```
final int[] numbers = {10, 20};
Object obj = new Object() {
public String toString() {
return "example: " + numbers[0];
}
};
```

提示 »» 要了解为什么，必须涉及一些底层机制。局部变量的生命周期往往比对象短，如方法调用后返回对象，局部变量生命周期就结束了，此时再通过对象尝试存取局部变量会发生错误，Java 的做法是采用传值。以上例而言，实际上会在匿名内部类的实例中，建立新的变量参考原来的对象。例如，尝试使用 JAD 反编译之后，可以看到：

```
int ai[] = {10, 20};
Object obj = new Object(ai) {
    public String toString() {
        return (new StringBuilder()).append("example: ").append(x[0]).toString();
    }
    final int x[];
    {
        x = ai;
    }
};
```

所以实际上也不能改变 x 的参考，为此，编译程序才会强制用户在局部变量加上 final。

在 JDK8 中，如果变量本身等效于 final 局部变量，也就是说，如果变量不会在匿名类中有重新指定的动作，就可以不加上 final 关键字，因此，图 7.8 的程序代码在 JDK8 中并不会发生编译错误。

7.2.3　使用 enum 枚举常数

在 7.2.1 节介绍过使用接口定义枚举常数的应用，当时定义了 play() 方法做示范：

```
...
    public static void play(int action) {
        switch(action) {
            case Action.STOP:
                System.out.println("播放停止动画");
```

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
                break;
            ...
            default:
                System.out.println("不支持此动作");
        }
    }
...
```

play()对于枚举常数的应用方式，问题在于参数接受的是 int 类型，这表示可以传入任何的 int 值，因此不得已地使用 default，以处理执行时期传入非定义范围的 int 值。

从 JDK5 之后新增了 **enum** 语法，可用于定义枚举常数。来看范例：

Enum Action.java

```
package cc.openhome;

public enum Action {
    STOP, RIGHT, LEFT, UP, DOWN
}
```

使用 enum 定义枚举常数，这是最简单的例子。实际上，enum 定义了特殊的类，继承自 **java.lang.Enum**，不过这是由编译程序处理，直接撰写程序继承 **Enum** 类会被编译程序拒绝。在编译过后，会产生 Action.class 文件，尝试反编译观察程序代码，可以了解 enum 枚举常数的部分细节：

```
public final class Action extends Enum {
    ...
    private Action(String s, int i) {
        super(s, i);
    }
    public static final Action STOP;
    public static final Action RIGHT;
    public static final Action LEFT;
    public static final Action UP;
    public static final Action DOWN;
    ...
    static {
        STOP = new Action("STOP", 0);
        RIGHT = new Action("RIGHT", 1);
        LEFT = new Action("LEFT", 2);
        UP = new Action("UP", 3);
        DOWN = new Action("DOWN", 4);
        ...
    }
}
```

可以看到，范例的 enum 定义的 Action 实际上是个类，而 **enum** 中列举的 **STOP**、**RIGHT**、**LEFT**、**UP**、**DOWN** 常数，实际上是 **public static final**，且为 **Action** 实例，用户无法撰写程序直接实例化 Action，因为构造函数权限设定为 **private**，只有在 Action 类中才可以实例化。

那么如何使用这个 Action 呢？用它来声明类型。例如：

Enum Game.java

```
package cc.openhome;

import static java.lang.System.out;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
public class Game {
    public static void main(String[] args) {
        play(Action.RIGHT); ←— ❶只能传入Action实例
        play(Action.UP);
    }

    public static void play(Action action) { ←— ❷声明为Action类型
        switch(action) {
            case STOP:     // 也就是Action.STOP ←— ❸列举Action实例
                out.println("播放停止动画");
                break;
            case RIGHT:    // 也就是Action.RIGHT
                out.println("播放向右动画");
                break;
            case LEFT:     // 也就是Action.LEFT
                out.println("播放向左动画");
                break;
            case UP:       // 也就是Action.UP
                out.println("播放向上动画");
                break;
            case DOWN:     // 也就是Action.DOWN
                out.println("播放向下动画");
                break;
        }
    }
}
```

在这个范例中，play()方法中的 action 参数声明为 Action 类型❷，所以只接受 Action 的实例，也就是只有 Action.STOP、Action.RIGHT、Action.LEFT、Action.UP、Action.DOWN 可以传入❶。不像 7.2.1 节中的 play()方法，可以传入任何 int 值，case 比较也只能列举 Action 实例❸。所以不用像前面的范例，必须使用 default 在执行时期检查，编译程序在编译时期会进行类型检查。

初学者要先了解以上对 enum 的使用，更多 enum 细节会在第 18 章再做说明。

7.3 重点复习

对于“定义行为”，可以使用 interface 关键字定义，接口中的方法不能操作，直接标示为 abstract，而且一定是 public。类要操作接口，必须使用 implements 关键字。在操作某接口时，对接口中定义的方法有两种处理方式，一是操作接口中定义的方法，二是再度将该方法标示为 abstract。

以 Java 的语意来说，继承会有“是一种”关系，操作接口则表示“拥有行为”，但不会有“是一种”的关系。对于接口多态语法的判断，方式是“右边是不是拥有左边的行为”或者“右边对象是不是操作了左边接口”。

类可以操作两个以上的类，也就是拥有两种以上的行为。类可以同时继承某个类，并操作某些接口。接口可以继承自另一个接口，也就是继承父接口行为，再在子接口中额外定义行为。

使用 interface 来定义抽象的行为外观，方法要声明为 public abstract，无须且不能有操作。为了方便，也可以省略 public abstract，编译程序会协助补齐。

可以使用接口枚举常数，只能定义为 public static final。为了方便，public static final

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

可以省略，编译程序会协助补齐。

如果有两个接口都定义了某方法，操作两个接口的类并不会有错误，照样通过编译，但在设计上要思考一下：两个接口都有定义的方法是否表示不同的行为？

接口可以继承别的接口，也可以同时继承两个以上的接口，同样也是使用 extends 关键字，这代表了继承父接口的行为。

如果有临时继承某个类或操作某个接口并建立实例的需求，而这类子类或接口操作类只使用一次，不需要为这些类定义名称，这时可以使用匿名内部类来解决这个需求。匿名内部类的语法为：

```
new 父类()|接口() {
    // 类本体操作
};
```

从 JDK5 之后新增了 enum 语法，可用于定义枚举常数。enum 定义了特殊的类，继承自 java.lang.Enum，不过这是由编译程序处理，直接撰写程序继承 Enum 类会被编译程序拒绝。

enum 定义了类，而 enum 中列举的常数，实际上是 public static final，且为枚举类型实例，无法撰写程序直接实例化枚举类型，因为构造函数权限设定为 private，只有类中才可以实例化。

7.4 课后练习

1. 针对 5.1 节设计的 CashCard 类，老板要你写个 CashCardService 类，其中有个 save() 方法，可以把每个 CashCard 实例的 number、balance 与 bonus 储存下来，有个 load() 方法，可以指定卡号加载已储存的 CashCard：

```
public void save(CashCard cashCard)
public CashCard load(String number)
```

可是老板也还没决定要存为文件，存到数据库，或存到另一台计算机？你怎么写 CashCardService 呢？

提示»» 搜索关键字 DAO。

2. 假设今天要开发一个动画编辑程序，每个画面为一帧(Frame)，数帧可组合为动画列表(Play List)，动画列表可由其他已完成动画清单组成，也可在动画列表与列表间加入个别帧，如何设计程序解决这个需求？

提示»» 搜索关键字“Composite 模式”。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 8

异常处理

学习目标

- 使用 try、catch 处理异常
- 认识异常继承架构
- 认识 throw、throws 的使用时机
- 运用 finally 关闭资源
- 使用自动关闭资源语法
- 认识 AutoCloseable 接口

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

8.1 语法与继承架构

程序中总有些因意想不到的状况所引发的错误，Java 中的错误也以对象方式呈现为 java.lang.Throwable 的各种子类实例。只要你能捕捉包装错误的对象，就可以针对该错误做一些处理，例如，试恢复正常流程、进行日志(Logging)记录、以某种形式提醒用户等。

8.1.1 使用 try、catch

来看一个简单的程序，用户可以连续输入整数，最后输入 0 结束后会显示输入数的平均值：

TryCatch Average.java

```
package cc.openhome;

import java.util.Scanner;

public class Average {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        double sum = 0;
        int count = 0;
        while(true) {
            int number = console.nextInt();
            if(number == 0) {
                break;
            }
            sum += number;
            count++;
        }
        System.out.printf("平均 %.2f%n", sum / count);
    }
}
```

如果用户正确地输入每个整数，程序会如预期地显示平均值：

```
10 20 30 40 0
平均 25.00
```

如果用户不小心输入错误，那就会出现奇怪的信息，例如第三个数输入为 3o，而不是 30：

```
10 20 3o 40 0
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:860)
    at java.base/java.util.Scanner.next(Scanner.java:1497)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2161)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2115)
    at cc.openhome/cc.openhome.Average.main(Average.java:11)
```

这段错误信息对除错是很有价值的，不过先看错误信息的第一行：

```
Exception in thread "main" java.util.InputMismatchException
```

Scanner 对象的 nextInt()方法，可以将用户输入的下一个字符串剖析为 int 值，如果出现

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

InputMismatchException 错误信息，表示不符合 Scanner 对象预期，因为 Scanner 对象预期下一个字符串本身要代表数字。

Java 中所有错误都会被打包为对象，如果愿意，可以尝试(try)捕捉(catch)代表错误的对象后做一些处理。例如：

TryCatch Average2.java

```
package cc.openhome;

import java.util.*;

public class Average2 {
    public static void main(String[] args) {
        try {
            Scanner console = new Scanner(System.in);
            double sum = 0;
            int count = 0;
            while (true) {
                int number = console.nextInt();
                if (number == 0) {
                    break;
                }
                sum += number;
                count++;
            }
            System.out.printf("平均 %.2f%n", sum / count);
        } catch (InputMismatchException ex) {
            System.out.println("必须输入整数");
        }
    }
}
```

这里使用了 try、catch 语法，JVM 会尝试执行 try 区块中的程序代码。如果发生错误，执行流程会跳离错误发生点，然后比较 catch 括号中声明的类型是否符合被抛出的错误对象类型，如果是的话，就执行 catch 区块中的程序代码。

如下所示是一个执行无误的范例：

```
10 20 30 40 0
平均 25.00
```

范例中如果 nextInt() 发生 InputMismatchException 错误，流程就会跳到类型声明为 InputMismatchException 的 catch 区块，执行完 catch 区块后，没有其他程序代码了，所以程序就结束了。如下所示是一个执行时输入有误的范例：

```
10 20 3o 40 0
必须输入整数
```

这个范例示范了如何运用 try、catch，在错误发生时显示更友好的错误信息。有时错误可以在捕捉处理之后，尝试恢复程序正常执行流程。例如：

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

TryCatch Average3.java

```
package cc.openhome;

import java.util.*;

public class Average3 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        double sum = 0;
        int count = 0;
        while (true) {
            try {
                int number = console.nextInt();
                if (number == 0) {
                    break;
                }
                sum += number;
                count++;
            } catch (InputMismatchException ex) {
                System.out.printf("略过非整数输入：%s%n", console.next());
            }
        }
        System.out.printf("平均 %.2f%n", sum / count);
    }
}
```

如果 nextInt()发生了 InputMismatchException 错误，执行流程就会跳到 catch 区块，执行完 catch 区块之后，由于还在 while 循环中，所以还可继续下一个循环流程。

如下是一个输入错误时的结果示范，对于正确的整数输入予以加总，对于错误的输入则显示略过，最后显示平均值。

```
10 20 3o 40 0
略过非整数输入：3o
平均 23.33
```

不过就 Java 在异常处理的设计上，并不鼓励捕捉 InputMismatchException，原因在介绍异常继承架构时会进行说明。

8.1.2 异常继承架构

在先前的 Average 范例中，虽然没有撰写 try、catch 语句，照样可以编译执行。初学者往往不理解的是，如图 8.1 所示撰写，编译为何会错误？

要解决这个错误信息有两种方式，一是使用 try、catch 打包 System.in.read()，二是在 main()方法旁声明 throws java.io.IOException。简单来说，编译程序认为调用 System.in.read()时有可能发生错误，要求你一定要在程序中明确处理错误。例如，如下撰写就可以通过编译：

```
try {
    int ch = System.in.read();
} catch(java.io.IOException ex) {
    ex.printStackTrace() ;
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Average 范例与这里的例子，程序都有可能发生错误，为什么编译程序只要求这里的范例，一定要处理错误呢？要了解这个问题，得先了解那些错误对象的继承架构，如图 8.2 所示。

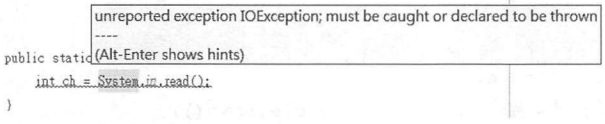


图 8.1　为什么一定要处理 java.io.IOException？

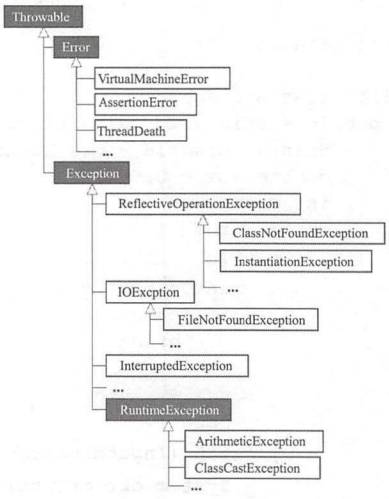


图 8.2　Throwable 继承架构图

首先要了解错误会被包装为对象，这些对象都是可抛出的(稍后介绍 throw 语法，就会了解如何抛出错误对象)，因此设计错误对象都继承自 **java.lang.Throwable** 类，Throwable 定义了取得错误信息、堆栈追踪(Stack Trace)等方法，它有两个子类：**java.lang.Error** 与 **java.lang.Exception**。

Error 与其子类实例代表严重系统错误，如硬件层面错误、JVM 错误或内存不足等问题。虽然也可以使用 try、catch 来处理 Error 对象，但并不建议。在发生严重系统错误时，Java 应用程序本身是无力回复的。举例来说，若 JVM 所需内存不足，如何撰写程序要求操作系统给予 JVM 更多内存呢？在 **Error** 对象抛出时，基本上不用处理，任其传播至 JVM 为止，或者是最多留下日志信息。

提示»» 如果抛出了 Throwable 对象，而程序中没有任何 catch 捕捉到错误对象，最后由 JVM 捕捉到的话，那 JVM 基本处理就是显示错误对象打包的信息并中断程序。

程序设计本身的错误，建议使用 **Exception** 或其子类实例来表现，所以通常称错误处理为异常处理(Exception Handling)。

单就语法与继承架构上来说，如果某个方法声明会抛出 Throwable 或子类实例，只要不是属于 **Error**、**java.lang.RuntimeException** 或其子类实例，就必须明确使用 try、catch 语法加以处理，或者用 throws 声明这个方法会抛出异常，否则会编译失败。

例如，先前调用 System.in.read()时，in 其实是 System 的静态成员，其类型为 java.io.InputStream，如果查询 API 文件，可以看到 InputStream 的 read()方法声明，如图 8.3 所示。

从图 8.3 可看到，IOException 是 Exception 的直接子类，所以编译程序要求明确使用语法加以处理。**Exception** 或其子对象，但非属于 **RuntimeException** 或其子对象，称为受检异常(Checked Exception)。受谁检查？受编译程序检查。受检异常存在之目的，在于 API 设计者在实现某方法时，某些条件成立时会引发错误，而且认为调用方法的客户端有能力处理错误，要求编译程序提醒客户端必须明确处理错误，不然不可通过编译，API 客户端无权选择要不要处理。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

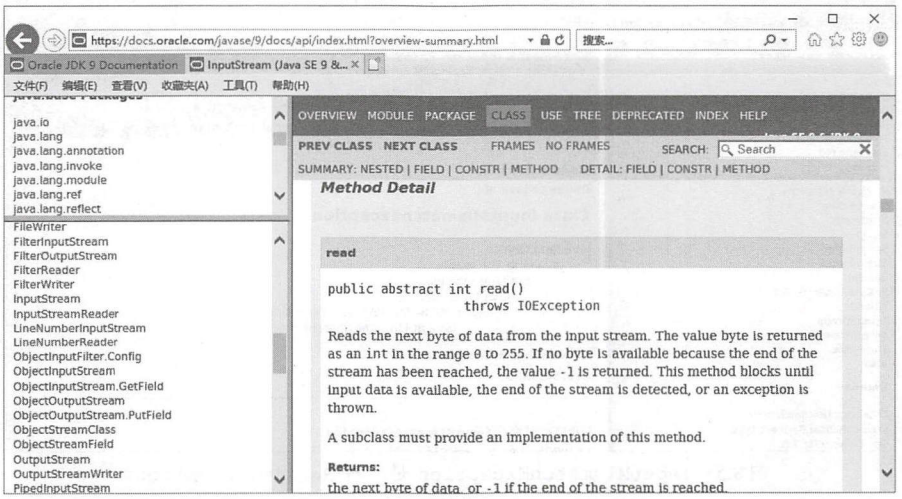


图 8.3 read()声明会抛出 IOException

提示>>> 图 8.2 中的 ReflectiveOperationException 是 JDK7 之后新增的类，JDK6 之前 ClassNotFoundException 等类是直接继承自 Exception 类。

属于 **RuntimeException** 衍生出来的类实例，在代表 API 设计者实现某方法时，某些条件成立时会引发错误，而且认为 API 客户端应该在调用方法前做好检查，以避免引发错误，之所以命名为执行时期异常，是因为编译程序不会强迫一定得在语法上加以处理，也称为非受检异常(Unchecked Exception)。

例如使用数组时，若存取超出索引就会抛出 ArrayIndexOutOfBoundsException，但编译程序并没有强迫你在语法上加以处理。这是因为 ArrayIndexOutOfBoundsException 是一种 RuntimeException，可以在 API 文件的开头找到继承架构图，如图 8.4 所示。

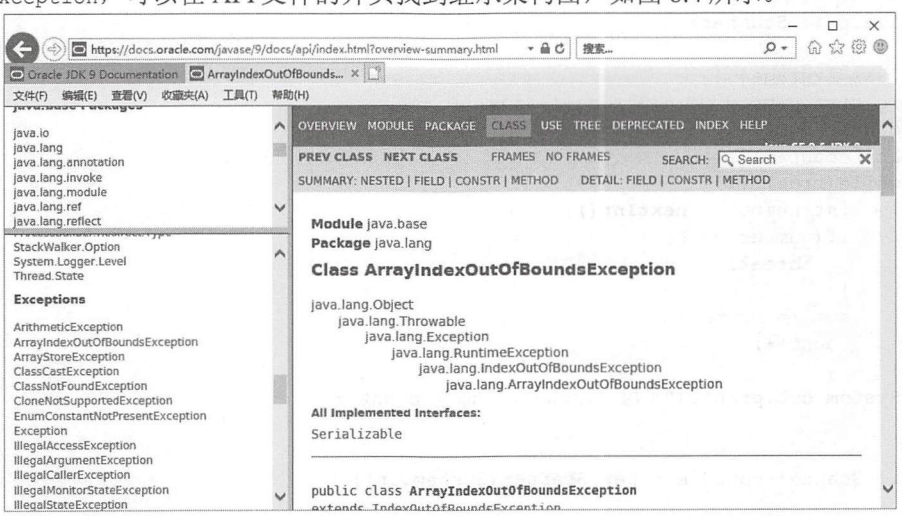


图 8.4 ArrayIndexOutOfBoundsException 是一种 RuntimeException

例如在 Average 范例中，用户输入是否正确并非事先可以得知，因此 InputMismatchException 设计为一种 RuntimeException，如图 8.5 所示。

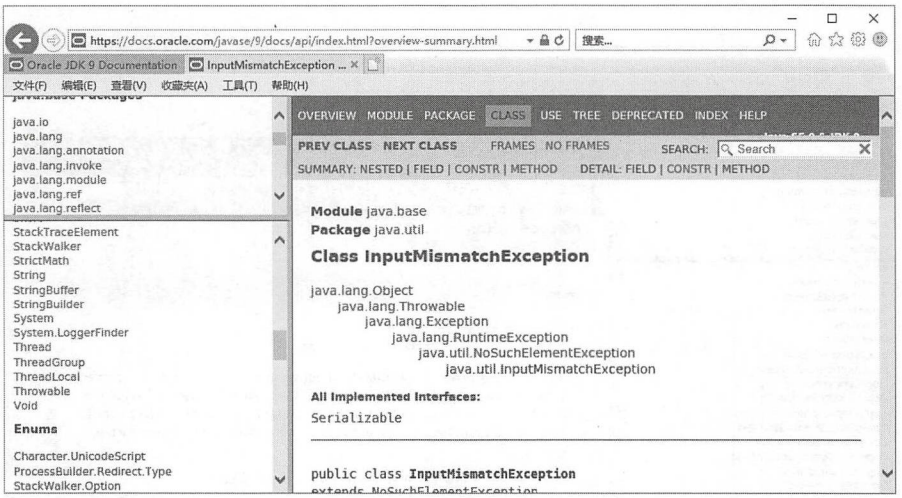


图 8.5 InputMismatchException 是一种 RuntimeException

Java 对于 RuntimeException 的态度是，这是因漏洞而引发，也就是 API 客户端调用方法前没有做好前置检查才会引发，客户端应该修改程序，使得调用方法时不会发生错误，如果真要以 try、catch 处理，建议是日志或呈现友好信息，例如之前的 Average2 范例的做法就是个例子。

虽然有些小题大作，不过前面的 Average3 范例若要避免出现 InputMismatchException，应该是取得用户的字符串输入之后，检查是否为数字格式，若是，再转换为 int 整数，若格式不对就提醒用户进行正确格式输入，例如：

TryCatch Average4.java

```
package cc.openhome;

import java.util.Scanner;

public class Average4 {
    public static void main(String[] args) {
        double sum = 0;
        int count = 0;
        while(true) {
            int number = nextInt();
            if(number == 0) {
                break;
            }
            sum += number;
            count++;
        }
        System.out.printf("平均 %.2f%n", sum / count);
    }

    static Scanner console = new Scanner(System.in);

    static int nextInt() {
        String input = console.next();
        while(!input.matches("\\d*")) {
            System.out.println("请输入数字");
            input = console.next();
```

```
        }
        return Integer.parseInt(input);
    }
}
```

上例的 nextInt()方法中，使用了 Scanner 的 next()方法来取得用户输入的下个字符串，如果字符串不是数字格式，就会提示用户输入数字，String 的 matches()方法中设定了"\\d*"，这是规则表示式(Regular Expression)，表示检查字符串中的字符是不是数字，若是则 matches()会返回 true，规则表示式在第 15 章还会介绍。

除了了解 Error 与 Exception 的区别，以及 Exception 与 RuntimeException 的分别之外，使用 try、catch 捕捉异常对象时也要注意，如果父类异常对象在子类异常对象前被捕捉，则 **catch** 子类异常对象的区块将永远不会被执行，编译程序会检查出这个错误，如图 8.6 所示。

```
try {
    System.in.read();
} catch(Exception ex
    ex.printStackTra
} catch(IOException ex) {
    ex.printStackTrace();
}
```

exception IOException has already been caught

(Alt-Enter shows hints)

图 8.6 了解异常继承架构是必要的

要完成这个程序的编译，必须更改异常对象捕捉的顺序。例如：

```
try {
    System.in.read();
} catch(IOException ex) {
    ex.printStackTrace();
} catch(Exception ex) {
    ex.printStackTrace();
}
```

数个类型的 catch 区块经常在做相同的事情，这种情况常发生在某些异常都要进行日志记录的时候。例如：

```
try {
    做一些事...
} catch(IOException e) {
    e.printStackTrace();
} catch(InterruptedException e) {
    e.printStackTrace();
} catch(ClassCastException e) {
    e.printStackTrace();
}
```

然而 catch 异常后的区块内容重复了，撰写时不仅无趣且对维护并没有帮助。从 JDK7 开始，可以使用多重捕捉(Multi-catch)语法：

```
try {
    做一些事...
} catch(IOException | InterruptedException | ClassCastException e) {
    e.printStackTrace();
}
```


非卖品，仅供非商业用途或交流学习使用

这样的撰写方式简洁许多，catch 区块会在发生 IOException、InterruptedException 或 ClassCastException 时执行，不过仍得注意异常的继承。catch 括号中列出的异常不得有继承关系，否则会发生编译错误，如图 8.7 所示。

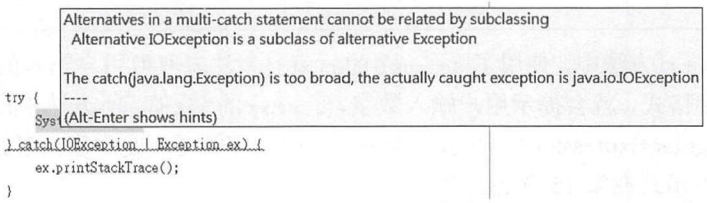


图 8.7　多重捕捉时也得注意异常继承架构

8.1.3　要抓还是要抛

假设今天你受命开发一个链接库，其中有个功能是读取纯文本文档，并以字符串返回文档中所有文字，你也许会这么撰写：

```
public class FileUtil {
    public static String readFile(String name) {
        StringBuilder text = new StringBuilder();
        try {
            Scanner console = new Scanner(new FileInputStream(name));
            while(console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        }
        return text.toString();
    }
}
```

虽然还没正式介绍到 Java 中如何存取文档，不过 4.1.2 节曾经介绍过，Scanner 创建时可以给予 InputStream 实例，而 FileInputStream 可指定文档名来开启与读取文档内容，是 InputStream 的子类，因此可作为 Scanner 创建之用。由于创建 FileInputStream 时会抛出 FileNotFoundException，根据目前学到的异常处理语法，你会捕捉 FileNotFoundException 并在控制台中显示错误信息。

控制台？等一下！老板有说这个链接库会用在文本模式中吗？如果这个链接库是用在 Web 网站上，发生错误时显示在控制台上，Web 用户怎么会看得到？你开发的是链接库，异常发生时如何处理，只有链接库用户才知道，直接在 catch 中写死处理异常或输出错误信息的方式，并不符合需求。

如果方法设计流程中发生异常，而你设计时并没有充足的信息知道该如何处理(例如不知道链接库会用在什么环境)，那么可以抛出异常，让调用方法的客户端来处理。例如：

```
public class FileUtil {
    public static String readFile(String name)
                                        throws FileNotFoundException {
```


非卖品，仅供非商业用途或交流学习使用

```
            StringBuilder text = new StringBuilder();
            Scanner console = new Scanner(new FileInputStream(name));
            while(console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
            return text.toString();
        }
    }
```

在操作对象的过程中如果会抛出受检异常，但目前环境信息不足以处理异常，无法使用 try、catch 处理时，可由方法的客户端依据当时调用的环境信息进行处理。为了告诉编译程序这个事实，必须使用 **throws** 声明此方法会抛出的异常类型或父类型，编译程序才会通过编译。

抛出受检异常，表示你认为调用方法的客户端有能力且应该处理异常，throws 声明部分会是 API 操作接口的一部分，客户端不用查看原始码，从 API 文件上就能直接得知，该方法可能抛出哪些异常。

如果你认为客户端调用方法的时机不当引发了某个错误，希望客户端准备好前置条件，再来调用方法，这时可以抛出非受检异常让客户端得知此情况。如果是非受检异常，编译程序不会要求明确使用 try、catch 或在方法上使用 throws 声明，因为 Java 的设计上认为，非受检异常是程序设计不当引发的漏洞，异常应自动往外传播，不应使用 try、catch 来尝试处理，而应改善程序逻辑来避免引发错误。

实际上在异常发生时，可使用 try、catch 处理当时环境可进行的异常处理，当时环境下无法决定如何处理的部分，可以抛出由调用方法的客户端处理。如果想先处理部分事项再抛出，可以如下撰写：

TryCatch FileUtil.java

```
package cc.openhome;

import java.io.*;
import java.util.Scanner;

public class FileUtil {
    public static String readFile(String name) throws FileNotFoundException {
        StringBuilder text = new StringBuilder();
        try {
            Scanner console = new Scanner(new FileInputStream(name));
            while(console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
            throw ex;
        }
        return text.toString();
    }
}
```

❶ 声明方法中会抛出异常

❷ 执行时抛出异常

在 catch 区块进行完部分错误处理之后，可以使用 **throw**(注意不是 throws)将异常再抛出❷，

非卖品，仅供非商业用途或交流学习使用

实际上，你可以在任何流程中抛出异常，不一定要在 catch 区块中。在流程中抛出异常，就直接跳离原有的流程，可以抛出受检或非受检异常。记住！如果抛出的是受检异常，表示你认为客户端有能力且应处理异常，此时必须在方法上使用 throws 声明❶；如果抛出的异常是非受检异常，表示你认为客户端调用方法的时机出错了，抛出异常是要求客户端修正这个漏洞再来调用方法，所以不使用 throws 声明。

如果原先有个方法操作是这样的：

```
public static void doSome(String arg)throws FileNotFoundException, EOFException {
    try {
        if("one".equals(arg)) {
            throw new FileNotFoundException();
        } else {
            throw new EOFException();
        }
    } catch(FileNotFoundException ex) {
        ex.printStackTrace();
        throw ex;
    } catch(EOFException ex) {
        ex.printStackTrace();
        throw ex;
    }
}
```

FileNotFoundException 与 EOFException 都是一种 IOException，而且 catch 后都做相同的事，想要使用 IOException 来 catch 这两种类型的异常，以下的写法在 JDK7 之前都会出错，如图 8.8 所示。

```
static void doSome(String arg)
        throws FileNotFoundException, EOFException {
    try {
        if ("one".equals(arg)) {
            throw new FileNotFoundException();
        } else {
            throw new EOFException();
        }
    } catch (IOE
        ex.print
        throw ex;
    }
}
```

unreported exception IOException; must be caught or declared to be thrown
----
(Alt-Enter shows hints)

图 8.8　多重捕捉时也得注意异常继承架构

在这个程序片段中，虽然实际上捕捉到的一定是 FileNotFoundException 与 EOFException 实例，方法上也使用 throws 予以声明了，但 JDK7 出现之前的编译程序没那么聪明，因而出现编译错误。在 JDK7 之后，编译程序对于重新抛出的异常类型可以更精准判断(More-Precise-Rethrow)，因此上面的程序片段，在 JDK7 之后不会再出现编译错误。

如果使用继承时，父类某个方法声明 throws 某些异常，子类重新定义该方法时可以：

- 不声明 throws 任何异常。
- throws 父类该方法中声明的某些异常。
- throws 父类该方法中声明异常的子类。

非卖品，仅供非商业用途或交流学习使用

但是不可以：

- throws 父类方法中未声明的其他异常。
- throws 父类方法中声明异常的父类。

8.1.4 贴心还是造成麻烦

异常处理的本意是，在程序错误发生时，能够有明确的方式通知 API 客户端，让客户端采取进一步的动作修正错误。就撰写本书的时间点来说，Java 是唯一采用受检异常(Checked Exception)的语言，这有两个目的：一是文件化，受检异常声明会是 API 操作接口的一部分，客户端只要查阅文件，就可以知道方法可能会引发哪些异常，并事先加以处理，而这是 API 设计者决定是否抛出受检异常时的考虑之一；二是提供编译程序信息，让编译程序能够在编译时期就检查出 API 客户端没有处理异常。

问题是有些错误发生而引发异常时，你根本无力处理，例如使用 JDBC 撰写数据库联机程序时，经常要处理的 java.sql.SQLException 是受检异常，如果异常的发生原因是数据库联机异常，而联机异常的原因是由于实体线路问题，那么无论如何你都不可能使用 try、catch 处理这种情况。

8.1.3 节中提到，错误发生时，如果当时情境并没有足够的信息让你处理异常，你可以就现有信息处理完异常后，重新抛出异常。你也许会这么写：

```
public Customer getCustomer(String id) throws SQLException {
    ...
}
```

看起来似乎没有问题，但假设这个方法是在整个应用程序底层被调用，在某个 SQLException 发生时，最好的方法是将异常浮现至用户画面呈现，例如网页技术，将错误信息在网页上显示出来给管理人员。

为了让异常往上浮现，你也许会选择在每个方法调用上都声明 throws SQLException，但前面假设，这个方法的调用是在整个应用程序的底层，这样的做法也许会造成许多程序代码的修改(更别说要重新编译了)，另一个问题是，如果你根本无权修改应用程序的其他部分，这样的做法显然行不通。

受检异常本意良好，有助于程序设计人员注意到异常的可能性并加以处理，但在应用程序规模增大时，会逐渐对维护造成困难，上述情况不一定是你自定义 API 时发生，也可能是在底层引入了一个会抛出受检异常的 API 而发生类似情况。

重新抛出异常时，除了将捕捉到的异常直接抛出，也可以考虑为应用程序自定义专属异常类别，让异常更能表现应用程序特有的错误信息。在自定义异常类别时，可以继承 Throwable、Error 或 Exception 的相关子类，通常建议继承自 Exception 或其子类，如果不是继承自 Error 或 RuntimeException，那么就会是受检异常。

```
public class CustomizedException extends Exception { // 自定义受检异常
    ...
}
```

错误发生时，如果当时情境没有足够的信息让你处理异常，你可以就现有信息处理完异常后，重新抛出异常。既然你已经针对错误做了某些处理，那么也就可以考虑自定义异常，用以

非卖品，仅供非商业用途或交流学习使用

更精确地表示出未处理的错误。如果认为调用 API 的客户端应当有能力处理未处理的错误，那就自定义受检异常、填入适当错误信息并重新抛出，并在方法上使用 throws 加以声明；如果认为调用 API 的客户端没有准备好就调用了方法，才会造成还有未处理的错误，那就自定义非受检异常、填入适当错误信息并重新抛出。

```
public class CustomizedException extends RuntimeException { // 自定义非受检异常
    ...
}
```

一个基本的例子是这样的：

```
try {
    ...
} catch(SomeException ex) {
    // 做些可行的处理
    // 也许是 Logging 之类的
    throw new CustomizedException("error message...");      // Checked 或 Unchecked?
}
```

类似地，如果流程中要抛出异常，也要思考一下，这是客户端可以处理的异常吗？还是客户端没有准备好前置条件就调用方法才引发的异常？

```
if(someCondition) {
    throw new CustomizedException("error message"); // Checked 或 Unchecked?
}
```

无论如何，Java 采用了受检异常的做法，Java 的标准 API 似乎也打算一直这么区分下去，只是受检异常让开发人员无从选择，会由编译程序强制性要求处理，确实会在设计上造成麻烦，因而有些开发者在设计链接库时，干脆就选择完全使用非受检异常，一些会封装应用程序底层行为的框架，如 Spring 或 Hibernate，就选择了让异常体系是非受检异常。例如 Spring 中的 DataAccessException，或者是 Hibernate 3 中的 HibernateException，它们选择给予开发人员较大的弹性来面对异常(这也要求开发人员有更多的经验)。

随着应用程序的演化，异常也可以考虑演化，也许一开始是设计为受检异常，然而随着应用程序堆栈的加深，受检异常老是一层一层往外声明抛出造成麻烦时，这也许代表了原先认为客户端可处理的异常，每一层客户端实际上都无力处理了。每层客户端都无力处理的异常，也许该视为一种漏洞，也许客户端在呼叫时都该准备好前置条件再行调用，以避免引发错误，这时将受检异常演化为非受检异常，就有其必要。

实际上确实有这类例子，Hibernate 2 中的 HibernateException 是受检异常，然而 Hibernate 3 中的 HibernateException 变成了非受检异常。

然而，即使不用面对受检异常与非受检异常的区别，开发者仍然必须思考，这是客户端可以处理的异常吗？还是客户端没有准备好前置条件就调用方法才引发的异常？

8.1.5 认识堆栈追踪

在多重方法调用下，异常发生点可能是在某个方法之中，若想得知异常发生的根源，以及多重方法调用下异常的堆栈传播，可以利用异常对象自动收集的堆栈追踪(Stack Trace)来取得相关信息。

非卖品，仅供非商业用途或交流学习使用

查看堆栈追踪最简单的方法，就是直接调用异常对象的 **printStackTrace()**。例如：

TryCatch StackTraceDemo.java

```
package cc.openhome;

public class StackTraceDemo {
    public static void main(String[] args) {
        try {
            c();
        } catch(NullPointerException ex) {
            ex.printStackTrace();
        }
    }

    static void c() {
        b();
    }

    static void b() {
        a();
    }

    static String a() {
        String text = null;
        return text.toUpperCase();
    }
}
```

在这个范例程序中，c()方法调用 b()方法，b()方法调用 a()方法，而 a()方法中会因 text 参考至 null，而后试图调用 toUpperCase()引发 NullPointerException。假设事先并不知道这个调用的顺序(也许你是在使用一个链接库)，当异常发生而被捕捉后，可以调用 printStackTrace()在控制台显示堆栈追踪，如图 8.9 所示。

```
java.lang.NullPointerException
    at cc.openhome.StackTraceDemo.a(StackTraceDemo.java:22)
    at cc.openhome.StackTraceDemo.b(StackTraceDemo.java:17)
    at cc.openhome.StackTraceDemo.c(StackTraceDemo.java:13)
    at cc.openhome.StackTraceDemo.main(StackTraceDemo.java:6)
```

图 8.9 异常堆栈追踪

在堆栈追踪信息中显示了异常类型，最顶层是异常的根源，以下是调用方法的顺序，程序代码行数是对应于当初的程序原始码，如果使用 IDE，单击行数就会直接开启原始码并跳至对应行数(如果原始码存在的话)。printStackTrace()还可以接受 PrintStream、PrintWriter 的版本，可以将堆栈追踪信息以指定方式至输出目的地(例如文档)。

提示>>> 编译位码文档时，默认会记录原始码行数信息等除错信息，在使用 javac 编译时指定 -g:none 自变量就不会记录除错信息，编译出来的位码文档容量会比较小。

如果想要取得个别的堆栈追踪元素进行处理，则可以使用 getStackTrace()，这会返回 StackTraceElement 数组，数组中索引 0 为异常根源的相关信息，之后为各方法调用中的信息，可

非卖品，仅供非商业用途或交流学习使用

以使用 StackTraceElement 的 getClassName()、getFileName()、getLineNumber()、getMethodName()，或者是 JDK9 新增的 getModuleName()、getModuleVersion()、getClassLoaderName()等方法取得对应的信息。

提示 »» 在 Throwable 与 Thread 上都定义有 getStackTrace()方法，可以主动使用 new Throwable()或 Thread.currentThread()取得 Throwable 或 Thread 实例后，调用 getStackTrace()方法以取得 StackTraceElement 数组，而 JDK9 为了更便于访问堆栈追踪，还新增了 Stack-Walking API，这部分内容会在第 15 章进一步介绍。

要善用堆栈追踪，前提是程序代码中不可有私吞异常的行为，例如在捕捉异常后什么都不做：

```
try {
    ...
} catch(SomeException ex) {
    // 什么也没有，绝对不要这么做!
}
```

这样的程序代码会对应用程序维护造成严重伤害，因为异常信息会完全中止，之后调用此片段程序代码的客户端，完全不知道发生了什么事，造成除错异常困难，甚至找不出错误根源。

另一种对应用程序维护会有伤害的方式，就是对异常做了不适当的处理，或显示了不正确的信息。例如，有时是由于某个异常层级下引发的异常类型很多：

```
try {
    ...
} catch(FileNotFoundException ex) {
    做一些处理
} catch(EOFException ex) {
    做一些处理
}
```

有些程序设计人员为了省麻烦，或因为经常处理找不到文档的错误，因而写成这样：

```
try {
    ...
} catch(IOException ex) {
    System.out.println("找不到文档");
}
```

这类的程序代码在项目中很常见，假以时日或者是别人使用程序时，真的发生了 EOFException(或其他原因导致了 IOException 或其子类型异常)，但错误信息却会一直显示找不到文档，因而误导了除错的方向。

在使用 throw 重抛异常时，异常的追踪堆栈起点，仍是异常的发生根源，而不是重抛异常的地方。例如：

TryCatch StackTraceDemo2.java

```
ppackage cc.openhome;

public class StackTraceDemo2 {
    public static void main(String[] args) {
```


非卖品，仅供非商业用途或交流学习使用

```
        try {
            c();
        } catch(NullPointerException ex) {
            ex.printStackTrace();
        }
    }

    static void c() {
        try {
            b();
        } catch(NullPointerException ex) {
            ex.printStackTrace();
            throw ex;
        }
    }
    static void b() {
        a();
    }

    static String a() {
        String text = null;
        return text.toUpperCase();
    }
}
```

执行这个程序，会发生以下的异常堆栈信息，可看到两次都是显示相同的堆栈信息：

```
java.lang.NullPointerException
    at cc.openhome/cc.openhome.StackTraceDemo2.a(StackTraceDemo2.java:28)
    at cc.openhome/cc.openhome.StackTraceDemo2.b(StackTraceDemo2.java:23)
    at cc.openhome/cc.openhome.StackTraceDemo2.c(StackTraceDemo2.java:14)
    at cc.openhome/cc.openhome.StackTraceDemo2.main(StackTraceDemo2.java:6)
java.lang.NullPointerException
    at cc.openhome/cc.openhome.StackTraceDemo2.a(StackTraceDemo2.java:28)
    at cc.openhome/cc.openhome.StackTraceDemo2.b(StackTraceDemo2.java:23)
    at cc.openhome/cc.openhome.StackTraceDemo2.c(StackTraceDemo2.java:14)
    at cc.openhome/cc.openhome.StackTraceDemo2.main(StackTraceDemo2.java:6)
```

如果想要让异常堆栈起点为重抛异常的地方，可以使用 **fillInStackTrace()**方法，这个方法会重新装填异常堆栈，将起点设为重抛异常的地方，并返回 Throwable 对象。例如：

TryCatch StackTraceDemo3.java

```
package cc.openhome;

public class StackTraceDemo3 {
    public static void main(String[] args) {
        try {
            c();
        } catch(NullPointerException ex) {
            ex.printStackTrace();
        }
    }

    static void c() {
        try {
            b();
```

```
        } catch(NullPointerException ex) {
            ex.printStackTrace();
            Throwable t = ex.fillInStackTrace();
            throw (NullPointerException) t;
        }
    }

    static void b() {
        a();
    }

    static String a() {
        String text = null;
        return text.toUpperCase();
    }
}
```

执行这个程序，会发生以下信息，可看到第二次显示堆栈追踪的起点，就是重抛异常的起点：

```
java.lang.NullPointerException
    at cc.openhome/cc.openhome.StackTraceDemo3.a(StackTraceDemo3.java:28)
    at cc.openhome/cc.openhome.StackTraceDemo3.b(StackTraceDemo3.java:23)
    at cc.openhome/cc.openhome.StackTraceDemo3.c(StackTraceDemo3.java:14)
    at cc.openhome/cc.openhome.StackTraceDemo3.main(StackTraceDemo3.java:6)
java.lang.NullPointerException
    at cc.openhome/cc.openhome.StackTraceDemo3.c(StackTraceDemo3.java:17)
    at cc.openhome/cc.openhome.StackTraceDemo3.main(StackTraceDemo3.java:6)
```

8.1.6 关于 assert

有时候，需求或设计时就可确认，在程序执行的某个时间点或某个情况下，一定是处于或不处于何种状态，若不是，则是个严重错误。在开发过程中发现这种严重错误，必须立即停止程序确认需求与设计。

在程序执行的某个时间点或某个情况下，必然处于或不处于何种状态，这是一种断言(Assertion)，例如某个时间点程序某个变量值一定是多少。断言的结果一定是成立或不成立，预期结果与实际程序状态相同时，断言成立，否则断言不成立。

Java 在 JDK 1.4 之后提供 **assert** 语法，有两种使用的语法：

```
assert boolean_expression;
assert boolean_expression : detail_expression;
```

`boolean_expression` 若为 `true`，则什么事都不会发生，如果为 `false`，则会发生 **`java.lang.AssertionError`**，此时若采取的是第二个语法，则会将 `detail_expression` 的结果显示出来，如果当中是个对象，则调用 `toString()` 显示文字描述结果。

断言功能是在 JDK 1.4 之后提供，由于使用 `assert` 作为关键字，为了避免 JDK 1.3 或更早版本程序使用 `assert` 作为变量导致名称冲突问题，在默认执行时不启动断言检查。如果要在执行时启动断言检查，可以在执行 `java` 指令时，指定**`-enableassertions`** 或是**`-ea`** 自变量。

那么何时该使用断言呢？一般有几个建议：

- 在断言客户端调用方法前，已经准备好某些前置条件(通常在 `private` 方法之中)。

- 在断言客户端调用方法后，具有方法承诺的结果。
- 在断言对象某个时间点下的状态。
- 使用断言取代批注。
- 断言程序流程中绝对不会执行到的程序代码部分。

以第 5 章的 CashCard 对象为例，来看看它的 charge() 方法原先设计如下：

```
...
    public void charge(int money) {
        if(money > 0) {
            if(money <= this.balance) {
                this.balance -= money;
            }
            else {
                out.println("钱不够啦！");
            }
        }
        else {
            out.println("扣负数？这不是叫我储值吗？");
        }
    }
...
```

原先的设计在错误发生时，直接在控制台中显示错误信息，通过适当地将 charge() 方法中的子流程封装为方法调用，并将错误信息以例外抛出，原程序可修改如下：

```
...
    public void charge(int money) throws InsufficientException {
        checkGreaterThanZero(money);
        checkBalance(money);
        this.balance -= money;

        // this.balance 不能是负数
    }

    private void checkGreaterThanZero(int money) {
        if(money < 0) {
            throw new IllegalArgumentException("扣负数？这不是叫我储值吗？");
        }
    }

    private void checkBalance(int money) throws InsufficientException {
        if(money > this.balance) {
            throw new InsufficientException("钱不够啦！", this.balance);
        }
    }
...
```

提示 »» CashCard 完整的程序修改，是本章的课后练习。

这里假设余额不足是商务流程上可处理的错误，因此让 InsufficientException 继承自

Exception成为受检例外，要求客户端呼叫时必须处理，而调用charge()方法时，本来就不该传入负数，因此checkGraterThanZero()会抛出非受检的IllegalArgumentException，这是种让错的程序看得出错，借由防御式程序设计(Defensive Programming)来实现速错(Fail Fast)概念。

提示>>> 防御式程序设计有些不好的名声，不过并不是做了防御式程序设计就不好，读者可以参考下面的文章《避免隐藏错误的防御性设计》：
http://openhome.cc/Gossip/Programmer/DefensiveProgramming.html

实际上，checkGreaterThanZero()是一种前置条件检查，如果程序上线后就不再需要这种检查的话，可以将之以assert取代，并在开发阶段使用-ea选项，而程序上线后取消该选项。charge()方法中使用了批注来提示方法调用后的对象状态必定不可以为负，这不如使用assert取代，会更有实质的效益：

```
...
    public void charge(int money) throws InsufficientException {
        assert money >= 0 : "扣负数？这不是叫我储值吗？";

        checkBalance(money);
        this.balance -= money;

        assert this.balance >= 0 : " this.balance 不能是负数";
    }

    private void checkBalance(int money) throws InsufficientException {
        if(money > this.balance) {
            throw new InsufficientException("钱不够啦！", this.balance);
        }
    }
...
```

另一个使用断言的时机，如7.2.1节的Game类中，一定不能有default的状况，也可以使用assert来取代：

```
...
    public static void play(int action) {
        switch(action) {
            case Action.STOP:
                out.println("播放停止动画");
                break;
            case Action.RIGHT:
                out.println("播放向右动画");
                break;
            case Action.LEFT:
                out.println("播放向左动画");
                break;
            case Action.UP:
                out.println("播放向上动画");
                break;
            case Action.DOWN:
                out.println("播放向下动画");
```

```
                break;
            default:
                assert false : "非定义的常数";
        }
    }
...
```

用户在使用 play()时，一定要使用 Action 定义的枚举常数，如果不是，就有可能执行到 default，若此情况发生视为开发时期的严重错误，所以直接 assert false，必然断言失败。

注意»» 断言是判定程序中的某个执行点必然是或不是某个状态，所以不能当作 if 之类的判断式来使用，assert 不应当作程序执行流程的一部分。

8.2 异常与资源管理

在程序中因错误而抛出异常时，原本的执行流程就会中断，抛出异常处之后的程序代码就不会被执行。如果程序开启了相关资源，使用完毕后你是否考虑到关闭资源呢？若因错误而抛出异常，你的设计是否还能正确地关闭资源呢？

8.2.1 使用 finally

老实说，8.1.3 节撰写的 FileUtil 范例并不是很正确，之后在学习输入/输出时会介绍到，如果创建 FileInputStream 实例就会开启文档，在不使用时，应该调用 close()关闭文档。FileUtil 中是通过 Scanner 搭配 FileInputStream 来读取文档，实际上 Scanner 对象有个 close()方法，可以关闭 Scanner 相关资源与搭配的 FileInputStream。

那么要何时关闭资源呢？如下撰写并不是很正确：

```
...
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    Scanner console = new Scanner(new FileInputStream(name));
    while (console.hasNext()) {
        text.append(console.nextLine())
            .append('\n');
    }
    console.close();
    return text.toString();
}
...
```

如果 scanner.close()前发生了任何异常，执行流程就会中断，因此 scanner.close()就可能不会执行，那么 Scanner 及搭配的 FileInputStream 就不会被关闭。

如果最后一定要执行关闭资源的动作，try、catch 语法还可以搭配 **finally**，无论 **try** 区块中有无发生异常，若撰写有 **finally** 区块，则 **finally** 区块一定会被执行。例如：

TryCatchFinally FileUtil.java

```
package cc.openhome;

import java.io.*;
import java.util.Scanner;

public class FileUtil {
    public static String readFile(String name) throws FileNotFoundException {
        StringBuilder text = new StringBuilder();
        Scanner console = null;
        try {
            console = new Scanner(new FileInputStream(name));
            while (console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
        } finally {
            if(console != null) {
                console.close();
            }
        }
        return text.toString();
    }
}
```

由于 finally 区块一定会被执行，这个范例中 scanner 原先是 null，若 FileInputStream 创建失败，则 scanner 就有可能还是 null，因此在 finally 区块中必须先检查 scanner 是否有参考对象，有的话才进一步调用 close() 方法，否则 scanner 参考至 null 又打算调用 close() 方法，反而会抛出 NullPointerException。

如果程序撰写的流程中先 **return** 了，而且也有 **finally** 区块，那 **finally** 区块会先执行完后，再将值返回。例如，下面这个范例会先显示 finally...再显示 1：

TryCatchFinally FinallyDemo.java

```
package cc.openhome;

public class FinallyDemo {
    public static void main(String[] args) {
        System.out.println(test(true));
    }

    static int test(boolean flag) {
        try {
            if(flag) {
                return 1;
            }
        } finally {
            System.out.println("finally...");
        }
        return 0;
    }
}
```

8.2.2 自动尝试关闭资源

经常地，在使用 try、finally 尝试关闭资源时，会发现程序撰写的流程是类似的，就如先前 FileUtil 示范的，你会先检查 scanner 是否为 null，再调用 close() 方法关闭 Scanner。在 JDK7 之后，新增了尝试关闭资源(Try-With-Resources)语法，来看它如何使用。

TryCatchFinally FileUtil2.java

```
package cc.openhome;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileUtil2 {
    public static String readFile(String name) throws FileNotFoundException {
        StringBuilder text = new StringBuilder();
        try(Scanner console = new Scanner(new FileInputStream(name))) {
            while (console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
        }
        return text.toString();
    }
}
```

正如程序示范的，想要尝试自动关闭资源的对象，是撰写在 try 之后的括号中，如果无须 catch 处理任何异常，可以不用撰写，也不用撰写 finally 自行尝试关闭资源。

JDK7 的尝试关闭资源语法也是个编译程序蜜糖，尝试反编译观察看看，有助于了解这个语法是否符合你的需求：

```
...
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    Scanner console = new Scanner(new FileInputStream(name));
    Throwable localThrowable2 = null;
    try {
        while (console.hasNext()) {
            text.append(console.nextLine())
                .append('\n');
        }
    } catch (Throwable localThrowable1) {                    // 尝试捕捉所有错误
        localThrowable2 = localThrowable1;
        throw localThrowable1;
    }
    finally {
        if (console != null) {  // 如果 console 参考至 Scanner 实例
            if (localThrowable2 != null) {                   // 若前面有 catch 到其他异常
                try {
                    console.close();                         // 尝试关闭 Scanner 实例
```

```
                } catch (Throwable x2) {                    // 万一关闭时发生错误
                    localThrowable2.addSuppressed(x2);      // 在原异常对象中记录
                }
            } else {
                console.close();            // 若前面没有发生任何异常，就直接关闭 Scanner
            }
        }
    }
    return text.toString();
}
...
```

重要的部分，直接在程序代码中以批注方式说明了。若一个异常被 **catch** 后的处理过程引发另一个异常，通常会抛出第一个异常作为响应。**addSuppressed()** 方法是 JDK7 在 `java.lang.Throwable` 中新增的方法，可将第二个异常记录在第一个异常之中，JDK7 中与之相对应的是 **getSuppressed()** 方法，可返回 `Throwable[]`，代表先前被 `addSuppressed()` 记录的各个异常对象。

在使用自动尝试关闭资源语法时，也可以搭配 `catch`。例如，在发生 `FileNotFoundException` 时显示堆栈追踪信息：

```
...
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    try(Scanner console = new Scanner(new FileInputStream(name))) {
        while (console.hasNext()) {
            text.append(console.nextLine());
            text.append('\n');
        }
    } catch(FileNotFoundException ex) {
        ex.printStackTrace();
        throw ex;
    }
    return text.toString();
}
...
```

使用 JAD 反编译后可以看到，实际上前一个反编译程序片段中粗体字部分，产生在另一个 `try`、`catch` 区块中：

```
...
public static String readFile(String name) throws FileNotFoundException {
    StringBuilder text = new StringBuilder();
    try(
        //这个区块中是自动尝试关闭资源语法展开后的程序代码
        Scanner console = new Scanner(new FileInputStream(name));
        Throwable localThrowable2 = null;
        try {
            while (console.hasNext()) {
                text.append(console.nextLine())
                    .append('\n');
            }
```


严禁网络传播本PDF，违者责任自负！

```
            } catch (Throwable localThrowable1) {
                localThrowable2 = localThrowable1;
                throw localThrowable1;
            }
            finally {
                if (console != null) {
                    if (localThrowable2 != null) {
                        try {
                            console.close();
                        } catch (Throwable x2) {
                            localThrowable2.addSuppressed(x2);
                        }
                    } else {
                        console.close();
                    }
                }
            }
        } catch(FileNotFoundException ex) {
            ex.printStackTrace();
            throw ex;
        }
        return text.toString();
    }
    ...
```

在使用自动尝试关闭资源语法时，并不影响你对特定异常的处理。实际上，自动尝试关闭资源语法也仅协助你关闭资源，而非用于处理异常。从反编译的程序代码中也可以看到，在使用尝试关闭资源语法时，不要试图自行撰写程序代码关闭资源，这样会造成重复调用 `close()` 方法，实际上语意也是如此，既然要自动关闭资源了，又何必自行撰写程序代码来关闭呢？

在 JDK8 或 JDK7 中，使用尝试关闭资源语法时，如同前面看到的范例那样，参考资源的变量一定得在 `try` 的括号中声明，例如在 JDK8 或 JDK7 中的编译错误：

```
public static String readFile(Scanner scanner) {
    StringBuilder text = new StringBuilder();
    try(scanner) {  // 在 JDK8 或 JDK7 中会编译错误，然而 JDK9 可以通过编译
        while (scanner.hasNext()) {
            text.append(scanner.nextLine());
            text.append('\n');
        }
    }
    return text.toString();
}
```

这会使得你在 JDK8 或 JDK7 中，必须在 `try` 的括号中再次声明变量，如 `try(Scanner console = scanner)`，以便通过编译。然而在 JDK9 中，只要变量是 **`final`** 或等效于 **`final`**，就可以通过编译。

8.2.3 java.lang.AutoCloseable 接口

JDK7 的尝试关闭资源语法可套用的对象，必须操作 **`java.lang.AutoCloseable`** 接口，这可

严禁网络传播本PDF，违者责任自负！

以在 API 文件上得知，如图 8.10 所示。

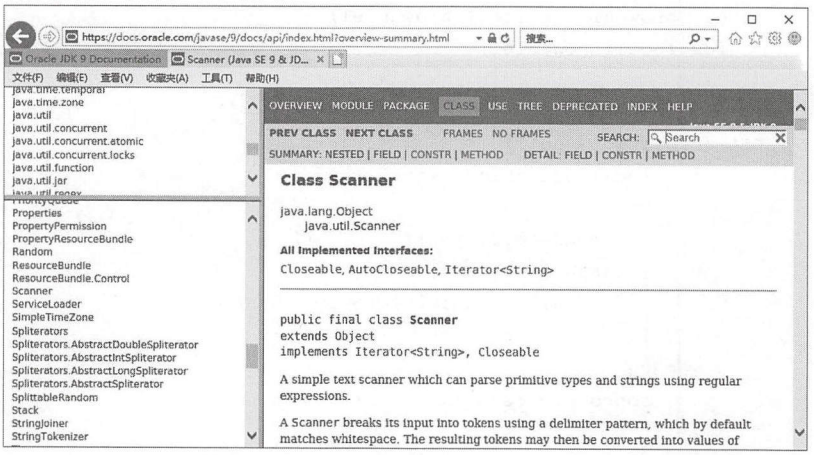


图 8.10　Scanner 操作了 AutoCloseable

AutoCloseable 是 JDK7 增加的接口，仅定义了 close() 方法：

```
package java.lang;
public interface AutoCloseable {
    void close() throws Exception;
}
```

所有继承 AutoCloseable 的子接口，或操作 AutoCloseable 的类，可在 AutoCloseable 的 API 文件上查询得知，如图 8.11 所示。

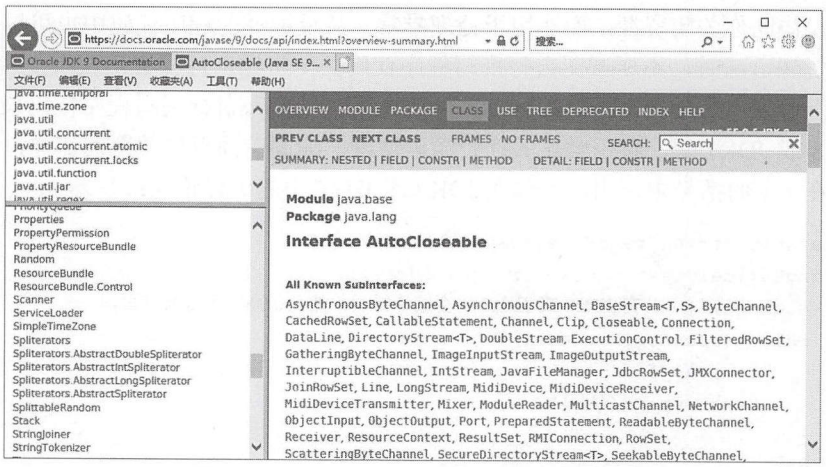


图 8.11　AutoCloseable 的子接口与操作类

只要操作 AutoCloseable 接口，就可以套用至尝试关闭资源语法。以下是个简单示范：

TryCatchFinally AutoClosableDemo.java

```
package cc.openhome;

public class AutoClosableDemo {
    public static void main(String[] args) {
        try(Resource res = new Resource()) {
            res.doSome();
        } catch(Exception ex) {
```


严禁网络传播本PDF，违者责任自负！

```
            ex.printStackTrace();
        }
    }
}

class Resource implements AutoCloseable {
    void doSome() {
        System.out.println("做一些事");
    }
    @Override
    public void close() throws Exception {
        System.out.println("资源被关闭");
    }
}
```

执行结果如下：

```
做一些事
资源被关闭
```

尝试关闭资源语法也可以同时关闭两个以上的对象资源，中间以分号分隔。来看看以下的范例，哪个对象资源会先被关闭呢？

TryCatchFinally AutoClosableDemo2.java

```
package cc.openhome;

import static java.lang.System.out;

public class AutoClosableDemo2 {
    public static void main(String[] args) {
        try(ResourceSome some = new ResourceSome();
            ResourceOther other = new ResourceOther()) {
            some.doSome();
            other.doOther();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

class ResourceSome implements AutoCloseable {
    void doSome() {
        out.println("做一些事");
    }
    @Override
    public void close() throws Exception {
        out.println("资源 Some 被关闭");
    }
}

class ResourceOther implements AutoCloseable {
    void doOther() {
        out.println("做其他事");
    }
    @Override
    public void close() throws Exception {
        out.println("资源 Other 被关闭");
    }
}
```


严禁网络传播本PDF，违者责任自负！

在 try 的括号中，越后面撰写的对象资源会越早被关闭。执行结果如下，ResourceOther 实例会先被关闭，然后再关闭 ResourceSome 实例：

```
做一些事
做其他事
资源 Other 被关闭
资源 Some 被关闭
```

可以实际观察反编译之后的程序代码，会发现每个 AutoCloseable 对象，都独立使用一个 try、catch、finally，try 括号中越后面撰写的对象，会是在越内层的 try、catch、finally 中：

```
...
try {
    ResourceSome some = new ResourceSome();
    Throwable localThrowable3 = null;
    try {
        ResourceOther other = new ResourceOther();
        Throwable localThrowable4 = null;
        try {
            some.doSome();
            other.doOther();
        } catch (Throwable localThrowable1) {
            localThrowable4 = localThrowable1;
            throw localThrowable1;
        } finally {  // 处理 ResourceOther 的关闭
            if (localThrowable4 != null) {
                try {
                    other.close();
                } catch (Throwable x2) {
                    localThrowable4.addSuppressed(x2);
                }
            } else {
                other.close();
            }
        }
    } catch (Throwable localThrowable2) {
        localThrowable3 = localThrowable2;
        throw localThrowable2;
    } finally {  // 处理 ResourceSome 的关闭
        if (localThrowable3 != null) {
            try {
                some.close();
            } catch (Throwable x2) {
                localThrowable3.addSuppressed(x2);
            }
        } else {
            some.close();
        }
    }
} catch (Exception ex) {
    ex.printStackTrace();
}
```

ResourceOther 的 try、catch、finally 部分

ResourceSome 的 try、catch、finally 部分

严禁网络传播本PDF，违者责任自负！

8.3 重点复习

Java 中所有错误都会被包装为对象，如果你愿意，可以尝试(try)执行程序并捕捉(catch)代表错误的对象后做一些处理。使用了 try、catch 语法，JVM 会尝试执行 try 区块中的程序代码，如果发生错误，执行流程会跳离错误发生点，然后比对 catch 括号中声明的类型，是否符合被抛出的错误对象类型，如果是的话，就执行 catch 区块中的程序代码。

错误会被包装为对象，这些对象都是可抛出的，因此设计错误对象都继承自 java.lang.Throwable 类。Throwable 定义了取得错误信息、堆栈追踪(Stack Trace)等方法，它有两个子类：java.lang.Error 与 java.lang.Exception。

Error 与其子类实例代表严重系统错误，例如硬件层面错误、JVM 错误或内存不足等问题。虽然也可以使用 try、catch 来处理 Error 对象，但并不建议，在发生严重系统错误时，Java 应用程序本身是无力处理的。

如果抛出了 Throwable 对象，而程序中没有任何 catch 捕捉到错误对象，最后由 JVM 捕捉到的话，JVM 基本处理就是显示错误对象包装的信息并中断程序。

程序设计本身的错误，建议使用 Exception 或其子类实例来表现，所以通常称错误处理为异常处理(Exception Handling)。

单就语法与继承架构上来说，如果某个方法声明会抛出 Throwable 或子类实例，只要不是属于 Error 或 java.lang.RuntimeException 或其子类实例，就必须明确使用 try、catch 语法加以处理，或者在方法中用 throws 声明这个方法会抛出异常，否则会编译失败。

Exception 或其子对象，但并非属于 RuntimeException 或其子对象，称为受检异常(Checked Exception)。受谁检查？受编译程序检查！受检异常存在的目的，在于 API 设计者实现某方法时，某些条件成立时会引发错误，而且认为调用方法的客户端有能力处理错误，要求编译程序提醒客户端必须明确处理错误，不然不可通过编译，API 客户端无权选择要不要处理。

属于 RuntimeException 衍生出来的类实例，代表 API 设计者在实现某方法时，某些条件成立而引发错误，而且认为 API 客户端应该在调用方法前做好检查，以避免引发错误，之所以命名为执行时期异常，是因为编译程序不会强迫一定得在语法上加以处理，亦称为非受检异常(Unchecked Exception)。

如果父类异常对象在子类异常对象前被捕捉，则 catch 子类异常对象的区块将永远不会被执行，编译程序会检查出这个错误。从 JDK7 开始，可以使用多重捕捉(Multi-catch)语法，不过仍得注意异常的继承，catch 括号中列出的异常不得有继承关系，否则会发生编译错误。

在操作对象的过程中如果会抛出受检异常，但目前环境信息不足以处理异常，所以在无法使用 try、catch 处理时，可由方法的客户端依据当时调用的环境信息进行处理。为了告诉编译程序这个事实，必须使用 throws 声明此方法会抛出的异常类型或父类型，编译程序才会通过编译。

如果是非受检异常，原本就可以自行选择是否处理异常，因此不使用 try、catch 处理时也不用特别在方法上使用 throws 声明，不处理非受检异常时，异常会自动往外传播。

在 catch 区块进行完部分错误处理之后，可以使用 throw(注意不是 throws)将异常再抛出。

严禁网络传播本PDF，违者责任自负！

在 JDK7 中，编译程序对于重新抛出的异常类型可以更精准判断。

若想得知异常发生的根源，以及多重方法调用下异常的堆栈传播，可以利用异常对象自动收集的堆栈追踪来取得相关信息，例如调用异常对象的 printStackTrace()、getStackTrace()等方法。

要善用堆栈追踪，前提是程序代码中不可有私吞异常的行为、对异常做了不适当的处理，或显示了不正确的信息。

在使用 throw 重抛异常时，异常的追踪堆栈起点，仍是异常的发生根源，而不是重抛异常的地方。如果想要让异常堆栈起点为重抛异常的地方，可以使用 fillInStackTrace()，这个方法会重新装填异常堆栈，将起点设为重抛异常的地方，并返回 Throwable 对象。

无论 try 区块中有无发生异常，若撰写有 finally 区块，则 finally 区块一定会被执行。如果程序撰写的流程中先 return 了，而且也有 finally 区块，finally 区块会先执行完后，再将值返回。

在 JDK7 之后，新增了尝试关闭资源(Try-With-Resources)语法，想要尝试自动关闭资源的对象，是撰写在 try 之后的括号中。

若一个异常被 catch 后的处理过程引发另一个异常，通常会抛出第一个异常作为响应，addSuppressed()方法是 JDK7 在 java.lang.Throwable 中新增的方法，可将第二个异常记录在第一个异常之中，JDK7 中与之相对应的是 getSuppressed()方法，可返回 Throwable[]，代表先前被 addSuppressed()记录的各个异常对象。

JDK7 的尝试关闭资源语法可套用的对象，必须操作 java.lang.AutoCloseable 接口，这是 JDK7 新增的接口。尝试关闭资源语法也可以同时关闭两个以上的对象资源，只要中间以分号分隔。在 try 的括号中，越后面撰写的对象资源会越早被关闭。

在 JDK9 中，只要变量执行了 java.lang.AutoCloseable 接口的实例，而且是 final 或等效于 final，在使用尝试关闭资源语法时，就可以放在 try 的括号中通过编译。

8.4 课后练习

1. 针对 5.2 节设计的 CashCard 类，store()与 charge()方法传入负数时，错误信息只显示在文本模式下，这当然不是正确的方式，请修改 store()与 charge()方法传入负数时，抛出 IllegalArgumentException。

2. 续上题，修改 CashCard 类，在点数或余额不足时，请抛出以下异常，其中 number 表示剩余点数或余额：

```
package cc.openhome.virtual;
public class InsufficientException extends Exception {
    private int number;
    public InsufficientException(String message, int remain) {
        super(message);
        this.number = remain;
    }
    public int getNumber() {
        return number;
    }
}
```

3. 续上题，修改 CashCard 类的 store()与 charge()方法，改以 assert 来断言不能传入负数。

严禁网络传播本PDF，违者责任自负！

Chapter 9

Collection 与 Map

学习目标

- 认识 Collection 与 Map 架构
- 使用 Collection 与 Map 操作对象
- 对收集的对象进行排序
- 简介 Lambda 表达式
- 简介泛型语法

严禁网络传播本PDF，违者责任自负！

9.1 使用 Collection 收集对象

程序中常有收集对象的需求，就目前为止，你学过可以收集对象的方式就是使用 Object 数组，而 6.2.5 节曾自行开发过 ArrayList 类，封装了自动增长 Object 数组长度等行为。在 Java SE 中，其实就提供了数个收集对象的类，你可以直接取用这些类，而不用重新打造类似的 API。

9.1.1 认识 Collection 架构

Java SE 提供了满足各种需求的 API，在使用这些 API 前，建议先了解其继承与接口操作架构，才能了解何时该采用哪个类，以及类之间如何彼此合作，而不会沦于死背 API 或抄写范例的窘境。

针对收集对象的需求，Java SE 提供了 Collection API，其接口继承架构如图 9.1 所示。

收集对象的行为，如新增对象的 add() 方法、移除对象的 remove() 方法等，都定义在 **java.util.Collection** 中。既然可以收集对象，也能逐一取得对象，这就是 **java.lang.Iterable** 定义的行为，它定义了 iterator() 方法返回 **java.util.Iterator** 操作对象，可以逐一取得收集的对象。其详细操作方式，稍后再做说明。

收集对象的共同行为定义在 Collection 中，然而收集对象会有不同的需求。如果希望收集时记录每个对象的索引顺序，并可依索引取回对象，这样的行为定义在 **java.util.List** 接口中。如果希望收集的对象不重复，具有集合的行为，由 **java.util.Set** 定义。如果希望收集对象时以队列方式，收集的对象加入至尾端，取得对象时从前端，可以使用 **java.util.Queue**。如果希望对 **Queue** 的两端进行加入、移除等操作，则使用 **java.util.Deque**。

在收集对象时，会依需求使用不同的接口操作对象。举例来说，如果想要在收集时具有索引顺序，操作方式之一就是使用数组，而以数组操作 List 的就是 java.util.ArrayList。如图 9.2 所示，如果查看 API 文件，会发现有以下继承与接口操作架构。

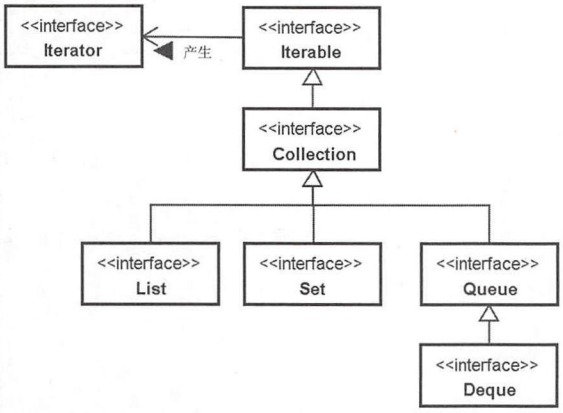


图 9.1 Collection API 接口继承架构

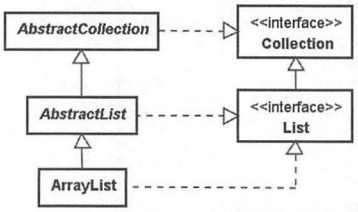


图 9.2 ArrayList 继承与接口操作架构

Java SE API 不仅提供许多已操作类，也考虑到用户自行扩充 API 的需求，以收集对象的基本行为来说，其提供 java.util.AbstractCollection 操作了 Collection 基本行为，java.util.AbstractList 操作了 List 基本行为，在必要时，还可以继承 AbstractCollection 操作自己的

严禁网络传播本PDF，违者责任自负！

Collection，继承 AbstractList 操作自己的 List，这会比直接操作 Collection 或 List 接口方便许多。

有时为了只表示感兴趣的界面或类，会简化继承与接口操作架构，如图 9.3 所示。

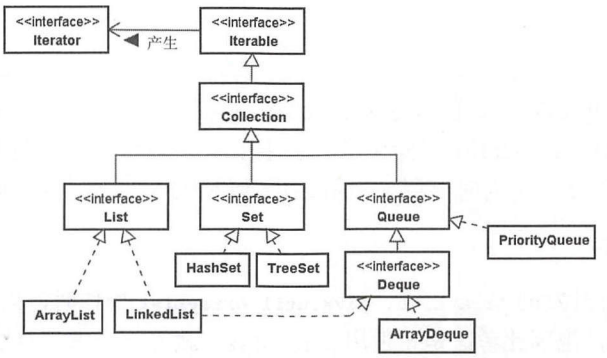


图 9.3　简化后的 Collection 继承与接口操作架构

这样的表示方式，可以更清楚地明了哪些类操作了哪个接口、继承了哪个类，或哪些接口又继承自哪个接口。至于详细的继承与操作架构，还是老话一句：可在 API 文件上查询。

9.1.2　具有索引的 List

List 是一种 Collection，作用是收集对象，并以索引方式保留收集的对象顺序，其操作类之一是 **java.util.ArrayList**，其操作原理大致如 6.2.5 节中的 ArrayList 范例。例如，可用 java.util.ArrayList 改写 6.2.5 节中的 Guest 类，而作用相同。

Collection Guest.java

```
package cc.openhome;

import java.util.*;
import static java.lang.System.out;

public class Guest {
    public static void main(String[] args) {
        List names = new ArrayList();  ←—— 使用 Java SE 的 List 与 ArrayList
        collectNameTo(names);
        out.println("访客名单：");
        printUpperCase(names);
    }

    static void collectNameTo(List names) {
        Scanner console = new Scanner(System.in);
        while(true) {
            out.print("访客名称：");
            String name = console.nextLine();
            if(name.equals("quit")) {
                break;
            }
            names.add(name);
        }
    }

    static void printUpperCase(List names) {
        for(int i = 0; i < names.size(); i++) {
```

```
            String name = (String) names.get(i); ←— 使用 get() 依索引取得收集的对象
            out.println(name.toUpperCase());
        }
    }
}
```

查看 API 文件，可发现 List 接口定义了 add()、remove()、set() 等许多依索引操作的方法。根据图 9.3，**java.util.LinkedList** 也操作了 List 接口。可以将上面的范例中的 ArrayList 换为 LinkedList，而结果不变，那何时该用 ArrayList、何时该用 LinkedList 呢？

1. ArrayList 特性

正如 6.2.5 节自行开发的 ArrayList、**java.util.ArrayList** 操作时，内部就是使用 **Object** 数组来保存收集的对象，也因此考虑是否使用 ArrayList，就等于考虑是否要使用到数组的特性。

数组在内存中会是连续的线性空间，根据索引随机存取时速度快，如果操作上有这类需求时，如排序，就可使用 ArrayList，得到较好的速度表现。

数组在内存中会是连续的线性空间，在需要调整索引顺序时，表现较差。例如在已收集 100 对象的 ArrayList 中，使用可指定索引的 add() 方法，将对象新增到索引 0 位置，那么原先索引 0 的对象必须调整至索引 1，索引 1 的对象必须调整至索引 2，索引 2 的对象必须调整至索引 3。依此类推，使用 ArrayList 做这类操作并不经济。

数组的长度固定也是要考虑的问题，在 ArrayList 内部数组长度不够时，会建立新数组，并将旧数组的参考指定给新数组，这也是必须耗费时间与内存的操作。为此，**ArrayList** 有个可指定容量(Capacity)的构造函数，如果大致知道将收集的对象范围，事先建立足够长度的内部数组，可以节省以上所描述的成本。

2. LinkedList 特性

LinkedList 在操作 List 接口时，采用了链接(Link)结构。若不是很了解何谓链接，可参考下面的 SimpleLinkedList 范例。

Collection SimpleLinkedList.java

```
package cc.openhome;

public class SimpleLinkedList {
    private class Node {
        Node(Object o) {
            this.o = o;
        }
        Object o;
        Node next;
    }

    private Node first; ←— ❷ 第一个节点

    public void add(Object elem) { ←— ❸ 新增 Node 封装对象，并由上一个 Node 的 next 参考
        Node node = new Node(elem);
        if(first == null) {
            first = node;
        }
        else {
```

❶ 将收集的对象用 Node 封装

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
            append(node);
        }
    }

    private void append(Node node) {
        Node last = first;
        while(last.next != null) {
            last = last.next;
        }
        last.next = node;
    }

    public int size() { ←— ❹ 访问所有 Node 并计数以取得长度
        int count = 0;
        Node last = first;
        while(last != null) {
            last = last.next;
            count++;
        }
        return count;
    }

    public Object get(int index) {
        checkSize(index);
        return findElemOf(index);
    }

    private void checkSize(int index) throws IndexOutOfBoundsException {
        int size = size();
        if(index >= size) {
            throw new IndexOutOfBoundsException(
                    String.format("Index: %d, Size: %d", index, size));
        }
    }

    private Object findElemOf(int index) { ←— ❺ 访问所有 Node 并计数以取得对应索引对象
        int count = 0;
        Node last = first;
        while(count < index) {
            last = last.next;
            count++;
        }
        return last.o;
    }
}
```

在 SimpleLinkedList 内部使用 Node 封装新增的对象❶，每次 add() 新增对象之后，将会形成链状结构❸。利用链接来收集对象，如图 9.4 所示。

所以每次 add() 对象时，才会建立新的 Node 来保存对象，不会事先耗费内存，若调用 size()❹，则从第一个对象❷，逐一参考下一个对象并计数，则可取得收集的对象长度。若想调用 get() 指定索引取得对象，则从第一个对象，逐一参考下一个对象并计数，取得指定索引的对象❺。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

可以看出，想要指定索引随机存取对象时，链接方式都得使用从第一个元素开始查找下一个元素的方式，效率较低，像排序就不适合使用链接操作的 List。想象一下，如果排序时，刚好必须将索引 0 与索引 10000 的元素调换，效率会不会好呢？

链接的每个元素会参考下一个元素，这有利于调整索引顺序。例如，在已收集 100 对象的 SimpleLinkedList 中，操作可指定索引的 add() 方法，将对象新增到索引 0 位置，则概念上如图 9.5 所示。

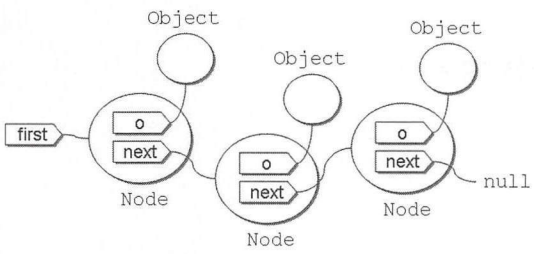


图 9.4　利用链接来收集对象

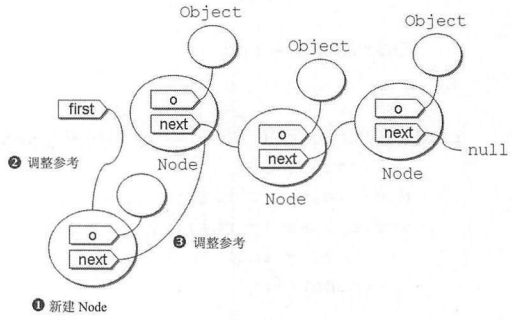


图 9.5　调整索引顺序所需动作较少

新增的对象将建立 Node 实例封装❶，而 first(或上一节点的 next)重新参考至新建的 Node 对象❷，新建 Node 的 next 则参考至下一 Node 对象❸。因此，若收集的对象经常会有变动索引的情况，也许考虑链接方式操作的 **List** 会比较好，如随时会有客户端登录或注销的客户端 List，使用 LinkedList 会有比较好的效率。

9.1.3　内容不重复的 Set

同样是收集对象，在收集过程中有相同对象，则不再重复收集，若有这类需求，可以使用 Set 接口的操作对象。例如，有一个字符串，当中有许多英文单词，你希望知道不重复的单词有几个，那么可以撰写如下程序：

Collection WordCount.java

```
package cc.openhome;

import java.util.*;

public class WordCount {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("请输入英文：");
        Set words = tokenSet(console.nextLine());
        System.out.printf("不重复单字有 %d 个：%s%n", words.size(), words);
    }

    static Set tokenSet(String line) {
        String[] tokens = line.split(" ");
        return new HashSet(Arrays.asList(tokens));
    }
}
```

❶ 显示收集的个数与字符串

❷根据空白切割出字符串

❸使用 HashSet 实现收集字符串

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

String 的 split() 方法，可以指定切割字符串的方式，在这里指定以空格切割，split() 会返回 String[]，包括切割的每个字符串❷，接着将 String[] 中的每个字符串加入 Set 的操作 HashSet 中❸。由于 Arrays.asList() 方法返回 List，而 List 是一种 Collection，因而可传给 HashSet 接收 Collection 实例的构造函数。由于 Set 的特性是不重复，因此若有相同单词，则不会再重复加入，最后只要调用 Set 的 size() 方法，就可以知道收集的字符串个数。HashSet 的 toString() 操作，则会包括收集的字符串❶。范例如下：

```
请输入英文: This is a dog that is a cat where is the student
不重复单词有 9 个: [that, cat, is, student, a, the, where, dog, This]
```

再来看以下范例：

Collection_Students.java

```
package cc.openhome;

import java.util.*;

class Student {
    private String name;
    private String number;
    Student(String name, String number) {
        this.name = name;
        this.number = number;
    }

    @Override
    public String toString()  {
        return String.format("(%s, %s)", name, number);
    }
}

public class Students {
    public static void main(String[] args) {
        Set students = new HashSet();
        students.add(new Student("Justin", "B835031"));
        students.add(new Student("Monica", "B835032"));
        students.add(new Student("Justin", "B835031"));
        System.out.println(students);
    }
}
```

程序中使用 Set 收集了 Student 对象，其中特意重复加入了相同的学生数据，然而在执行结果中看到，Set 并没有将重复的学生数据排除：

```
[(Monica, B835032), (Justin, B835031), (Justin, B835031)]
```

这是理所当然的结果，因为你并没有告诉 Set，什么样的 Student 实例才算是重复。以 HashSet 为例，会使用对象的 hashCode() 与 equals() 来判断对象是否相同。HashSet 的操作概念是，在内存中开设空间，每个空间会有个哈希编码(Hash Code)，如图 9.6 所示。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

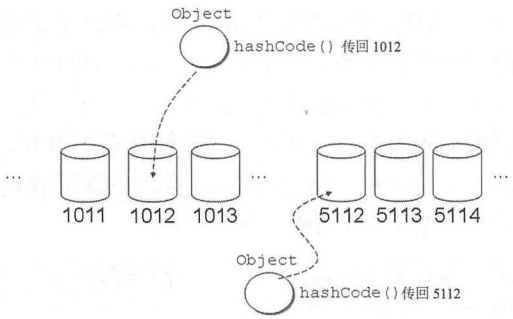


图 9.6 HashSet 操作概念

这些空间称为哈希桶(Hash Bucket)，如果对象要加入 HashSet，则会调用对象的 hashCode() 取得哈希码，并尝试放入对应号码的哈希桶中，如果哈希桶中没对象，直接放入，如图 9.6 所示；如果哈希桶中有对象呢？会再调用对象的 equals() 进行比较，如图 9.7 所示。

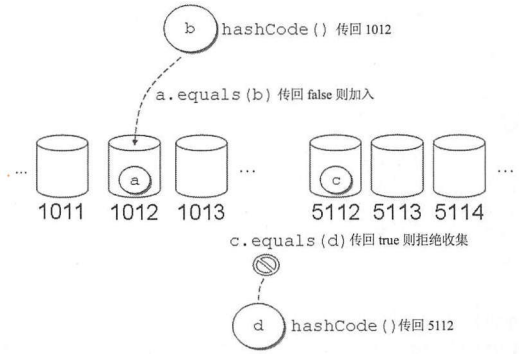


图 9.7 根据 equals() 与 hashCode() 判断要不要收集

如果同一个哈希桶中已有对象，调用该对象 equals() 与要加入的对象比较结果为 false，则表示两个对象非重复对象，可以收集；如果是 true，表示两个对象是重复对象，不予收集。

事实上不只有 HashSet，Java 中许多要判断对象是否重复时，都会调用 **hashCode()** 与 **equals()** 方法，因此规格书中建议，两个方法必须同时操作。以前面范例而言，若操作了 hashCode() 与 equals() 方法，则重复的 Student 将不会被收集。

Collection Students2.java

```
package cc.openhome;

import java.util.*;

class Student2 {
    private String name;
    private String number;
    Student2(String name, String number) {
        this.name = name;
        this.number = number;
    }

    // NetBeans 自动生成的 equals() 与 hashCode()
    // 就示范而言已经足够了
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    @Override
    public int hashCode() {
        // Objects 有 hash() 方法可以使用
        // 以下可以简化为 return Objects.hash(name, number);
        int hash = 7;
        hash = 47 * hash + Objects.hashCode(this.name);
        hash = 47 * hash + Objects.hashCode(this.number);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Student2 other = (Student2) obj;
        if (!Objects.equals(this.name, other.name)) {
            return false;
        }
        if (!Objects.equals(this.number, other.number)) {
            return false;
        }
        return true;
    }

    @Override
    public String toString()  {
        return String.format("(%s, %s)", name, number);
    }
}

public class Students2 {
    public static void main(String[] args) {
        Set students = new HashSet();
        students.add(new Student2("Justin", "B835031"));
        students.add(new Student2("Monica", "B835032"));
        students.add(new Student2("Justin", "B835031"));
        System.out.println(students);
    }
}
```

在这里定义学生的姓名与学号相同，表示为相同 `Student` 对象，`hashCode()` 则直接利用 `String` 的 `hashCode()` 再做运算(为 NetBeans 自动程序代码产生的结果)。在执行结果可看出不再收集重复的 `Student2` 对象：

```
[(Justin, B835031), (Monica, B835032)]
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

9.1.4 支持队列操作的 Queue

如果希望收集对象时以队列方式，收集的对象加入至尾端，取得对象时从前端，则可以使用 **Queue** 接口的操作对象。Queue 继承自 Collection，所以也具有 Collection 的 add()、remove()、element()等方法，然而 Queue 定义了自己的 **offer()**、**poll()**与 **peek()**等方法，最主要的差别在于，add()、remove()、element()等方法操作失败时会抛出异常，而 offer()、poll()与 peek()等方法操作失败时会返回特定值。

如果对象有操作 **Queue**，并打算以队列方式使用，且队列长度受限，通常建议使用 **offer()**、**poll()**与 **peek()**等方法。offer()方法用来在队列后端加入对象，成功会返回 true，失败则返回 false。poll()方法用来取出队列前端对象，若队列为空则返回 null。peek()用来取得(但不取出)队列前端对象，若队列为空则返回 null。

前面提过 **LinkedList**，它不仅操作了 List 接口，也操作了 **Queue** 的行为，所以可将 LinkedList 当作队列来使用。例如：

Collection RequestQueue.java

```
package cc.openhome;

import java.util.*;

interface Request {
    void execute();
}

public class RequestQueue {
    public static void main(String[] args) {
        Queue requests = new LinkedList();
        offerRequestTo(requests);
        process(requests);
    }

    static void offerRequestTo(Queue requests) {
        // 仿真将请求加入队列
        for (int i = 1; i < 6; i++) {
            Request request = new Request() {
                public void execute() {
                    System.out.printf("处理数据 %f%n", Math.random());
                }
            };
            requests.offer(request);
        }
    }
    // 处理队列中的请求
    static void process(Queue requests) {
        while(requests.peek() != null) {
            Request request = (Request) requests.poll();
            request.execute();
        }
    }
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

执行结果如下：

```
处理数据 0.302919
处理数据 0.616828
处理数据 0.589967
处理数据 0.475854
处理数据 0.274380
```

经常地，需要队列的前端与尾端进行操作，在前端加入对象与取出对象，在尾端加入对象与取出对象，Queue 的子接口 **Deque** 就定义了这类行为。Deque 中定义 addFirst()、removeFirst()、getFirst()、addLast()、removeLast()、getLast()等方法，操作失败时会抛出异常，而 offerFirst()、pollFirst()、peekFirst()、offerLast()、pollLast()、peekLast()等方法，操作失败时会返回特定值。

Queue 的行为与 Deque 的行为有所重复，有几个操作是等义的，如表 9.1 所示。

表 9.1 **Queue 与 Deque 等义方法**

Queue 方法	Deque 等义方法
add()	addLast()
offer()	offerLast()
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

java.util.ArrayDeque 操作了 Deque 接口，以下范例是使用 ArrayDeque 来操作容量有限的堆栈：

Collection Stack.java

```
package cc.openhome;

import java.util.*;
import static java.lang.System.out;

public class Stack {
    private Deque elems = new ArrayDeque();
    private int capacity;

    public Stack(int capacity) {
        this.capacity = capacity;
    }

    public boolean push(Object elem) {
        if(isFull()) {
            return false;
        }
        return elems.offerLast(elem);
    }

    private boolean isFull() {
        return elems.size() + 1 > capacity;
    }

    public Object pop() {
        return elems.pollLast();
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    }

    public Object peek() {
        return elems.peekLast();
    }

    public int size() {
        return elems.size();
    }

    public static void main(String[] args) {
        Stack stack = new Stack(5);
        stack.push("Justin");
        stack.push("Monica");
        stack.push("Irene");
        out.println(stack.pop());
        out.println(stack.pop());
        out.println(stack.pop());
    }
}
```

堆栈结构是先进后出，所以执行结果最后才显示 Justin：

```
Irene
Monica
Justin
```

提示>>> LinkedList 也操作了 Deque，不过就这里操作堆栈的范例而言，使用 ArrayDeque，速度上会有比较好的表现，你可以思考一下为什么。

9.1.5 使用泛型

在使用 Collection 收集对象时，由于事先不知道被收集对象的形态，因此在内部操作时，都是使用 Object 来参考被收集的对象，取回对象时也是以 Object 类型返回，原理可参考 6.2.5 节自行操作的 ArrayList，或 9.1.2 节操作的 SimpleLinkedList。

由于取回对象时会以 Object 类型返回，若想针对某类定义的行为操作时，必须告诉编译程序，让对象重新扮演该类型。例如：

```
List names = Arrays.asList("Justin", "Monica", "Irene") ;
String name = (String) names.get(0);
```

在 Collection 收集对象时，考虑到收集各种对象的需求，因而内部操作采用 Object 参考收集的对象，所以执行时期被收集的对象会失去形态信息，也因此取回对象之后，必须自行记得对象的真正类型，并在语法上告诉编译程序让对象重新扮演为自己的类型。

Collection 虽然可以收集各种对象，但实际上通常 Collection 中会收集同一种类型的对象，例如都是收集字符串对象。因此从 JDK5 之后，新增了泛型(Generics)语法，这样，用户在设计 API 时可以指定类或方法支持泛型，而使用 API 的客户端在语法上会更为简洁，并得到编译时期检查。

以 6.2.5 节自行操作的 ArrayList 为例，可加入泛型语法：

Collection ArrayList.java

```
package cc.openhome;

import java.util.Arrays;

public class ArrayList<E> {  ←— ❶ 此类支持泛型
    private Object[] elems;
    private int next;

    public ArrayList(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList() {
        this(16);
    }

    public void add(E e) {  ←— ❷加入的对象必须是客户端声明的 E 类型
        if(next == elems.length) {
            elems = Arrays.copyOf(elems, elems.length * 2);
        }
        elems[next++] = e;
    }

    public E get(int index) {  ←— ❸ 取回对象以客户端声明的 E 型态返回
        return (E) elems[index];
    }

    public int size() {
        return next;
    }
}
```

注意到范例中粗体字部分，首先类名称旁出现了角括号<E>，这表示此类支持泛型❶。实际加入 ArrayList 的对象会是客户端声明的 E 类型。E 只是一个类型代号(表示 Element)，也可以用 T、K、V 等代号。

由于使用<E>定义类型，在需要编译程序检查类型的地方，都可以使用 E，如 add() 方法必须检查传入的对象类型是 E❷，get() 方法必须转换为 E 类型❸。

使用泛型语法，会对设计 API 造成一些语法上的麻烦，但对客户端会多一些友好。例如：

```
...
ArrayList<String> names = new ArrayList<String>();
names.add("Justin");
names.add("Monica");
String name1 = names.get(0);
String name2 = names.get(1);
...
```

声明与建立对象时，可使用角括号告知编译程序，这个对象收集的都会是 String，而取回之后也会是 String，不再使用括号转换类型。如果实际上加入了不是 String 的内容会如何呢？如图 9.8 所示，编译程序会检查加入的类型。

```
ArrayList<String> names
names.add("Justin");
names.add("Monica");
names.add(new Integer(10));
```

incompatible types: Integer cannot be converted to String

(Alt-Enter shows hints)

图 9.8　编译程序会检查加入的类型

由于你告诉编译程序，这个 ArrayList 收集的对象都会是 String，若收集非 String 的对象，编译程序就会检查出这个错误。

Java 的 Collection API 都支持泛型语法，若在 API 文件上看到角括号，表示支持泛型语法，如图 9.9 所示。

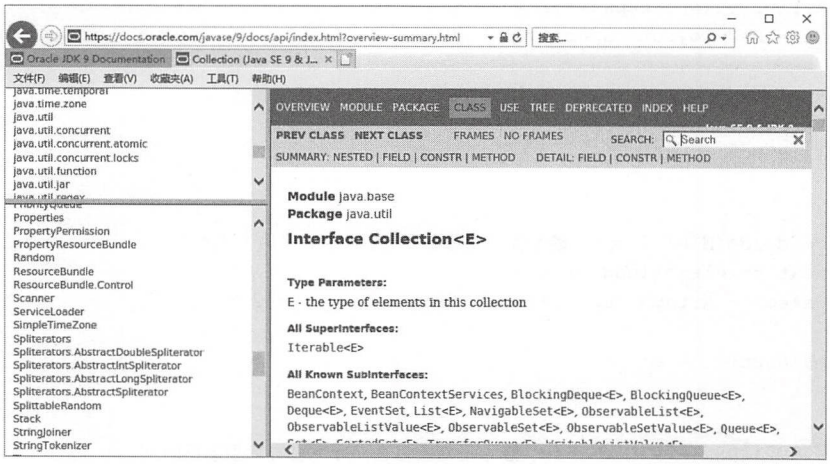


图 9.9　API 文件上的角括号代表支持泛型

这类 API 若在运用时，没有指定类型参数实际类型，程序代码中出现类型参数的地方，就会回归为使用 Object 类型。

以使用 java.util.List 为例，若要指定类型参数，可以这样声明：

```
...
List<String> words = new LinkedList<String>();
words.add("one");
String word = words.get(0);
...
```

实际上，泛型语法有一部分是编译程序蜜糖(一部分是记录在位码中的信息)。若反编译以上程序片段，可以看到还是展开为 JDK1.4 之前的写法：

```
...
LinkedList linkedlist = new LinkedList();
linkedlist.add("one");
String s = (String) linkedlist.get(0);
...
```

正因为展开后会有粗体字部分的语法，因而以下会编译错误：

```
List<String> words = new LinkedList<String>();
words.add("one");
Integer number = words.get(0);              // 编译错误
...
```

编译程序展开程序代码后，实际上会如下：

```
List words = new LinkedList();
words.add("one");
Integer number = (String) words.get(0);   // 编译错误
...
```

若接口支持泛型，在操作时也会比较方便。例如有如下接口声明：

```
...
public interface Comparator<T> {
    int compare(T o1, T o2);
    ...
}
```

这表示操作接口时，可以指定 T 代表的类型，而 compare() 就可以直接套用 T 类型。例如：

```
public class StringComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return -s1.compareTo(s2);
    }
}
```

如果不指定 T 的实际类型，那么 T 出现的位置就回归为使用 Object，就上例来说，实作上就会啰唆一些，例如：

```
public class StringComparator implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
    String s1 = (String) o1;
    String s2 = (String) o2;
        return -s1.compareTo(s2);
    }
}
```

再来看以下程序片段：

```
List<String> words = new LinkedList<String>();
```

你会不会觉得有点啰唆呢？明明声明 words 已经使用 List<String>告诉编译程序，words 参考的对象中，都会是 String 了，为什么创建 LinkedList 时，还要用 LinkedList<String>再告知呢？这个问题从 JDK7 之后有了点改善，这样撰写就可以了：

```
List<String> words = new LinkedList<>();
```

只要声明参考时有指定类型，那么创建对象时就不用再写类型了。

泛型也可以仅定义在方法上，最常见的是在静态方法上定义泛型，例如，原本有个 elemOf() 方法如下：

```
public static Object elemOf(Object[] objs, int index) {
    return objs[index];
}
```

如果有个 String[]的 args 要传给 elemOf()方法，取回索引 i 的 String 对象，那么得这么写：

```
String arg = (String) elemOf(args, i);
```


非卖品，仅供非商业用途或交流学习使用

若能将 elemOf() 设计为泛型方法，则：

```
public static <T> T elemOf(T[] objs, int index) {
    return objs[index];
}
```

如果包括这个 elemOf() 方法的类是 Util，那就可以使用 Util.<String>elemOf() 的方式，指定 T 的实际类型。事实上，如果编译程序可以自动从程序代码中，推断出 T 的实际类型，那么，就可以不用自行指定 T 的类型。例如：

```
String arg = elemOf(args, i);
```

泛型语法还有许多细节，就初学者而言，只要先知道以上的应用即可。更多泛型语法细节，将会在第 18 章介绍。

提示»» 适当地使用泛型语法，语法上可以简洁一些，编译程序也可以事先做类型检查，但泛型语法也可以用得很复杂(有人称为魔幻，这是个贴切的新形容词)，使用时以不影响可读性与维护性为主要考虑。

9.1.6 简介 Lambda 表达式

回顾一下 9.1.4 节中的 RequestQueue 范例，其中使用了匿名类语法来实现 Request 接口的实例如：

```
Request request = new Request() {
public void execute() {
       out.printf("处理数据 %f%n", Math.random());
   }
};
```

可以看到信息重复了，声明 request 变量时已经告知是 Request 类型，而建立 Request 实例的匿名类语法又写了一次，实际上 **Request** 接口只有一个方法必须实现，当这种情况发生时，在 JDK8 中可以使用 Lambda 表达式(Expression)如下撰写：

```
Request request = () -> out.printf("处理数据 %f%n", Math.random());
```

相对于匿名类语法来说，Lambda 表达式的语法省略了接口类型与方法名称，->左边是参数列，而右边是方法本身，编译程序可以由 **Request request** 的声明中得知语法上被省略的信息。

来看另一个例子，如果有个接口声明如下：

```
public interface IntegerFunction {
    Integer apply(Integer i);
}
```

类似地，可以使用匿名类来实现 IntegerFunction:

```
IntegerFunction doubleFunction = new IntegerFunction() {
    public Integer apply(Integer i) {
        return i * 2;
    }
}
```

然而语法上又会出现重复信息的问题，若改用 JDK8 的 Lambda 表达式，可以改写为以下较简洁的写法：

```
IntegerFunction doubleFunction = (Integer i) -> i * 2;
```


非卖品，仅供非商业用途或交流学习使用

实际上，编译程序可以从 IntegerFunction doubleFunction 中得知，你实现了 IntegerFunction 的实例。既然如此，编译程序应该也可以得知，参数 i 的类型是 Integer。那么，以下写法也是可行的：

```
IntegerFunction doubleFunction = (i) -> i * 2;
```

由于编译程序具备类型推断(Type Inference)的能力，让 Lambda 表达式可以更简洁地撰写，实际上，如果是单参数又无须写出参数类型的时候，() 也可以省略，因此上式还可以写为：

```
IntegerFunction doubleFunction = i -> i * 2;
```

使用 Lambda 表达式，编译程序在推断类型时，还可以用泛型声明的类型作为信息来源。例如有个接口声明如下：

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

若以匿名类语法来实现这个接口的话，会是这样：

```
Comparator<String> byLength = new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
};
```

重复的信息更多了，Comparator、String 与 compare 等都是重复的信息，虽然从 JDK9 开始，定义匿名类时，等号左边已经指定了泛型的实际类型，右边就可以省略类型声明：

```
Comparator<String> byLength = new Comparator<>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
};
```

如果改用 Lambda 语法来实现，加上编译程序的类型推断能力辅助，上例可以改写为更简洁的方式：

```
Comparator<String> byLength = (name1, name2) -> name1.length() - name2.length();
```

结合到目前已介绍的泛型与 Lambda 表达式，来改写 9.1.4 节中的 RequestQueue 范例，看看会不会简洁一些：

Collection RequestQueue2.java

```
package cc.openhome;

import java.util.*;

interface Request2 {
    void execute();
}

public class RequestQueue2 {
    public static void main(String[] args) {
        Queue<Request2> requests = new LinkedList<>();
        offerRequestTo(requests);
```


非卖品，仅供非商业用途或交流学习使用

```
        process(requests);
    }

    static void offerRequestTo(Queue<Request2> requests) {
        // 仿真将请求加入队列
        for (int i = 1; i < 6; i++) {
            requests.offer(
                () -> System.out.printf("处理数据 %f%n", Math.random())
            );
        }
    }
    // 处理队列中的请求
    static void process(Queue<Request2> requests) {
        while(requests.peek() != null) {
            Request2 request = requests.poll();
            request.execute();
        }
    }
}
```

虽然不鼓励使用Lambda表达式来写复杂的演算，不过若流程较为复杂，无法在一行Lambda表达式中写完时，可以使用区块{}符号包括演算流程。例如：

```
Request request = () -> {
out.printf("处理数据 %f%n", Math.random());

};
```

在Lambda表达式中使用区块时，如果方法必须返回值，在区块中就必须使用 return。例如：

```
IntegerFunction doubleFunction = i -> {
    return i * 2;
}
```

这里对于Lambda表达式只是简介，只是JDK8的Lambda特性中的一小部分，第12章会完整地介绍Lambda。运用目前已学会的Lambda表达式，读者在撰写 Collection 相关功能时，也可让程序代码本身更有表达能力，更容易理解。

9.1.7 Interable 与 Iterator

如果要写个 forEach()方法，可以显示 List 收集的所有对象，也许你会这么写：

```
...
    static void forEach(List list) {
        int size = list.size();
        for(int i = 0; i < size; i++) {
            out.println(list.get(i));
        }
    }
...
```

这个方法适用于所有操作 List 接口的对象，如 ArrayList、LinkedList 等。如果要写个 forEach()方法显示 Set 收集的所有对象，你该怎么写呢？在查看过 Set 的API文件后，你发现有个 toArray()方法，可以将 Set 收集的对象转为 Object[]返回，所以你可能会这么撰写：

```
...
    static void forEach(Set set) {
```


非卖品，仅供非商业用途或交流学习使用

```
        for(Object o : set.toArray()) {
            out.println(o);
        }
    }
...
```

这个方法适用于所有操作 Set 接口的对象，如 HashSet、TreeSet 等。如果现在要让你再操作一个 forEach()方法，可以显示 Queue 收集的所有对象，也许你会这么写：

```
...
    static void forEach(Queue queue) {
        while(queue.peek() != null) {
            out.println(queue.poll());
        }
    }
...
```

表面上看来好像没问题，不过 Queue 的 poll()方法会取出对象，当显示完 Queue 中所有对象，Queue 也空了，这并不是想要的结果，怎么办呢？

事实上，无论 List、Set 还是 Queue，都会有个 **iterator()**方法，这个方法在 JDK5 出现之前，定义在 **Collection** 接口中，而 List、Set、Queue 继承自 Collection，所以也都拥有 iterator()的行为。

iterator()方法会返回 **java.util.Iterator** 接口的操作对象，这个对象包括了 Collection 收集的所有对象，可以使用 Iterator 的 **hasNext()**看看有无下一个对象，若有的话，再使用 **next()**取得下一个对象。因此，无论 List、Set、Queue 还是其他 Collection，都可以使用以下的 forEach()来显示所收集的对象：

```
...
    static void forEach(Collection collection) {
        Iterator iterator = collection.iterator();
        while(iterator.hasNext()) {
            out.println(iterator.next());
        }
    }
...
```

在 JDK5 之后，原先定义在 Collection 中的 iterator()方法，提升至新的 java.lang.Iterable 父接口，因此在 JDK5 之后，可以使用以下的 forEach()方法显示收集的所有对象：

```
...
    static void forEach(Iterable iterable) {
        Iterator iterator = iterable.iterator();
        while(iterator.hasNext()) {
            out.println(iterator.next());
        }
    }
...
```

提示 »» 任何操作 Iterable 的对象，都可以使用这个 forEach()方法，而不一定要是 Collection。本章课后练习的操作题中，将要求你写个 IterableString 类，可以运用这个 forEach()方法逐一显示字符串中的字符。

非卖品，仅供非商业用途或交流学习使用

在JDK5之后有了增强式 **for** 循环，前面看到它运用在数组上，实际上，增强式 for 循环还可运用在操作 **Iterable** 接口的对象上。因此前面的 forEach()方法，用增强式 for 循环更加简化：

Collection ForEach.java

```
package cc.openhome;

import java.util.*;

public class ForEach {
    public static void main(String[] args) {
        List names = Arrays.asList("Justin", "Monica", "Irene"); ❶
        forEach(names); ❷
        forEach(new HashSet(names)); ❸
        forEach(new ArrayDeque(names)); ❹
    }

    static void forEach(Iterable iterable) {
        for(Object o : iterable) {
            System.out.println(o);
        }
    }
}
```

这里使用了 **java.util.Arrays** 的 static 方法 **asList()**，这个方法接受不定长度自变量，可将指定的自变量收集为 List❶。List 是一种 Iterable，可以使用 forEach()方法❷。HashSet 具有接受 Collection 的构造函数，List 是一种 Collection，可用来创建 HashSet，而 Set 是一种 Iterable，可使用 forEach()方法❸。同理，ArrayDeque 具有接受 Collection 的构造函数，List 是一种 Collection，所以可用来创建 ArrayDeque，Deque 是一种 Iterable，所以可使用 forEach()方法❹。

实际上增强式 **for** 循环是编译程序蜜糖，当运用在 Iterable 对象时，会展开为：

```
private static void forEach(Iterable iterable) {
    Object o;
    for(Iterator i$ = iterable.iterator();
                i$.hasNext();
System.out.println(o)) {
        o = i$.next();
    }
}
```

可以看到，实际上还是调用了 iterator()方法，运用返回的 Iterator 对象来迭代取得收集的对象。

如果使用JDK8，想要迭代对象还有新的选择，Iterable 新增了 forEach()方法，可以让你迭代对象进行指定处理：

```
List<String> names = Arrays.asList("Justin", "Monica", "Irene");
names.forEach(name -> out.println(name));
new HashSet(names).forEach(name -> out.println(name));
new ArrayDeque(names).forEach(name -> out.println(name));
```

Iterable 的 forEach()方法接受 java.util.function.Consumer<T>接口的实例，这个接口只有一个 accept(T t)方法必须实现，与其使用匿名类来实现，不如使用 Lambda 表示式来得更清楚，程序代码本身也更有表达能力。

非卖品，仅供非商业用途或交流学习使用

提示 »» 实际上，JDK8 的 Lambda 特性中还可以进行方法参考(Method Reference)，可让上面的程序代码更简洁地撰写成：

```
List<String> names = Arrays.asList("Justin", "Monica", "Irene");
names.forEach(out::println);
new HashSet(names).forEach(out::println);
new ArrayDeque(names).forEach(out::println);
```

如果你了解 JDK8 之前的接口，应该也知道，接口一旦新增了方法，所有实现接口的类都得操作该方法，现存各种 API 中实现 Iterable 接口的类太多了，这样不会造成这些类因为没有操作 forEach()而在 JDK8 上编译错误吗？不会的！因为 JDK8 改进了 interface 语法，允许接口定义默认方法(Default Method)，方法参考与默认方法都会在第 12 章详细介绍。

9.1.8 Comparable 与 Comparator

在收集对象之后，对对象进行排序是常用的动作，你不用亲自操作排序算法，`java.util.Collections` 提供 `sort()`方法。由于必须有索引才能进行排序，因此 `Collections` 的 `sort()`方法接受 `List` 操作对象。例如：

```
jshell> List numbers = Arrays.asList(10, 2, 3, 1, 9, 15, 4);
numbers ==> [10, 2, 3, 1, 9, 15, 4]

jshell> Collections.sort(numbers);
|  Warning:
|  unchecked method invocation: method sort in class java.util.Collections is applied to
given types
|    required: java.util.List<T>
|    found: java.util.List
|  Collections.sort(numbers);
|  ^----------------------^
|  Warning:
|  unchecked conversion
|    required: java.util.List<T>
|    found:    java.util.List
|  Collections.sort(numbers);
|                   ^-----^

jshell> numbers
numbers ==> [1, 2, 3, 4, 9, 10, 15]
```

提示 »» 在上面的 jshell 中看到了警告，这是因为使用了 API，而你没有声明实际类型，在编译时若遇到这类情况，实际上编译程序也会提出警告：

```
Note: Main.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

若编译时遇到 unchecked 警告，在 `javac` 指令后加上`-Xlint:unchecked` 自变量重新编译，就可以看到警告的来源细节。

如果是以下的范例呢？

Collection Sort2.java

```
package cc.openhome;

import java.util.*;
```


非卖品，仅供非商业用途或交流学习使用

```
class Account {
    private String name;
    private String number;
    private int balance;

    Account(String name, String number, int balance) {
        this.name = name;
        this.number = number;
        this.balance = balance;
    }

    @Override
    public String toString() {
        return String.format("Account(%s, %s, %d)", name, number, balance);
    }
}

public class Sort {
    public static void main(String[] args) {
        List accounts = Arrays.asList(
                new Account("Justin", "X1234", 1000),
                new Account("Monica", "X5678", 500),
                new Account("Irene", "X2468", 200)
        );
        Collections.sort(accounts);
        System.out.println(accounts);
    }
}
```

执行结果将会抛出 `ClassCastException`?

```
Exception in thread "main" java.lang.ClassCastException:
cc.openhome.Account cannot be cast to java.lang.Comparable
...
```

1. 操作 Comparable

要说原因，是因为你根本没告诉 `Collections` 的 `sort()` 方法，到底要根据 `Account` 的 `name`、`number` 或 `balance` 哪个进行排序，那它要怎么排？`Collections` 的 `sort()` 方法要求被排序的对象，必须操作 **`java.lang.Comparable`** 接口，这个接口有个 **`compareTo()`** 方法必须返回大于 0、等于 0 或小于 0 的数，作用为何？直接来看如何针对账户余额进行排序就可以了解：

Collection Sort2.java

```
package cc.openhome;

import java.util.*;

class Account2 implements Comparable<Account2> {
    private String name;
    private String number;
    private int balance;

    Account2(String name, String number, int balance) {
        this.name = name;
```


非卖品，仅供非商业用途或交流学习使用

```
        this.number = number;
        this.balance = balance;
    }

    @Override
    public String toString() {
        return String.format("Account2(%s, %s, %d)", name, number, balance);
    }

    @Override
    public int compareTo(Account2 other) {
        return this.balance - other.balance;
    }
}

public class Sort2 {
    public static void main(String[] args) {
        List accounts = Arrays.asList(
                new Account2("Justin", "X1234", 1000),
                new Account2("Monica", "X5678", 500),
                new Account2("Irene", "X2468", 200)
        );
        Collections.sort(accounts);
        System.out.println(accounts);
    }
}
```

Collections 的 sort()方法在取得 a 对象与 b 对象进行比较时，会先将 a 对象扮演(Cast)为 Comparable(也因此若对象没操作 Comparable，将会抛出 ClassCastException)，然后调用 **a.compareTo(b)**，如果 a 对象顺序上小于 b 对象则返回小于 0 的值，若顺序上相等则返回 0，若顺序上 a 大于 b 则返回大于 0 的值。因此，上面的范例，将会根据余额从小到大排列账户对象：

```
    [Account2(Irene, X2468, 200), Account2(Monica, X5678, 500), Account2(Justin, X1234,
1000)]
```

为何前面的 Sort 类中，可以直接对 Integer 进行排序呢？若查看 API 文件，可以发现 Integer 操作了 Comparable 接口，如图 9.10 所示。

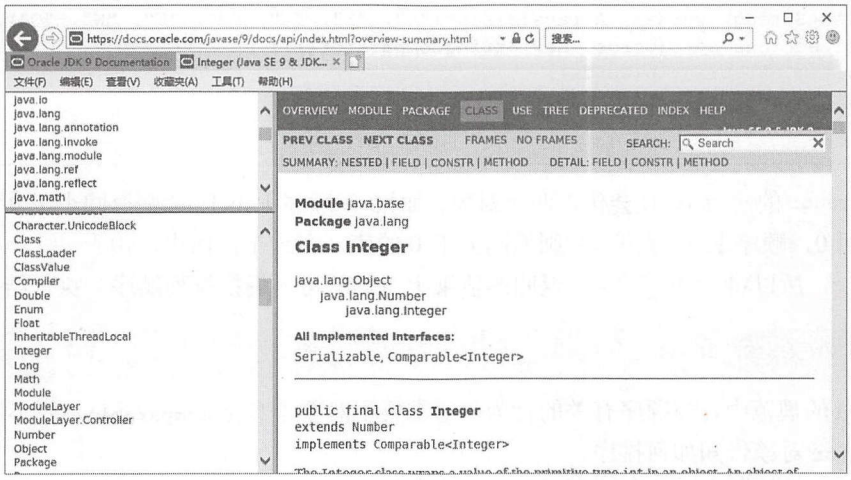


图 9.10　Integer 操作了 Comparable 接口

非卖品，仅供非商业用途或交流学习使用

2. 操作 Comparator

实际开发总是不断有意外，如果对象无法操作 Comparable 呢？也许你拿不到原始码，也许你不能修改原始码。举个例子来说，String 本身有操作 Comparable，所以可以如下排序：

```
jshell> List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
words ==> [B, X, A, M, F, W, O]

jshell> Collections.sort(words);

jshell> words;
words ==> [A, B, F, M, O, W, X]
```

如果你想要让排序结果反过来呢？修改 String.java？这个方法不可行吧。就算你修改后重新编译为 String.class 放回 JRE 中，也只有你的 JRE 可以用，这已经不是标准 API 了。继承 String 后再重新定义 compareTo()方法？也不可能，因为 String 声明为 final，不能被继承。

Collections 的 sort()方法有另一个重载版本，可接受 **java.util.Comparator** 接口的操作对象，如果使用这个版本，排序方式将根据 Comparator 的 **compare()**定义来决定。例如：

Collection Sort3.java

```
package cc.openhome;

import java.util.*;

class StringComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return -s1.compareTo(s2);
    }
}

public class Sort3 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
        Collections.sort(words, new StringComparator());
        System.out.println(words);
    }
}
```

Comparator 的 compare()会传入两个对象，如果 o1 顺序上小于 o2 则返回小于 0 的值，顺序相等则返回 0，顺序上 o1 大于 o2 则返回大于 0 的值。在这个范例中，由于 String 本身就是 Comparable，所以将 compareTo()返回的值乘上-1，就可以调换排列顺序。执行结果如下：

```
[X, W, O, M, F, B, A]
```

在 Java 的规范中，与顺序有关的行为，通常要不对象本身是 **Comparable**，要不就是另行指定 **Comparator** 对象告知如何排序。

如果想针对数组进行排序，可以使用 **java.util.Arrays** 的 **sort()**方法，如果查询 API 文

非卖品，仅供非商业用途或交流学习使用

件，会发现该方法针对对象排序时有两个版本：一个版本是你收集在数组中的对象必须是 Comparable(否则会抛出 ClassCastException)，另一个版本则可以传入 Comparator 指定排序方式。

Set 的操作类之一 **java.util.TreeSet**，不仅拥有收集不重复对象的能力，还可用红黑树方式排序收集的对象，条件就是收集的对象必须是 Comparable(否则会抛出 ClassCastException)，或者是在创建 TreeSet 时指定 Comparator 对象。

Queue 的操作类之一 **java.util.PriorityQueue** 也是，收集至 PriorityQueue 的对象，会根据用户指定的优先权来决定对象在队列中的顺序，优先权的告知，要不就是对象必须是 Comparable(否则会抛出 ClassCastException)，或者是创建 PriorityQueue 时指定 Comparator 对象。

对了！刚刚才简介过 Lambda 语法，Comparator 接口需要操作的只有一个 compare()方法，因此上面的范例可以使用 Lambda 语法来让它更简洁一些：

```
jshell> List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
words ==> [B, X, A, M, F, W, O]

jshell> Collections.sort(words, (s1, s2) -> -s1.compareTo(s2));

jshell> words;
words ==> [X, W, O, M, F, B, A]
```

实际上，JDK8 在 List 上增加了 sort()方法，可接受 Comparator 实例来指定排序方式，因此还可以写成：

```
jshell> List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
words ==> [B, X, A, M, F, W, O]

jshell> words.sort((s1, s2) -> -s1.compareTo(s2));

jshell> words;
words ==> [X, W, O, M, F, B, A]
```

提示》》 如果只是想使用 String 的 compareTo()方法，通过 JDK8 的方法参考特性，实际上还可以写得更简洁：

```
List<String> words = Arrays.asList("B", "X", "A", "M", "F", "W", "O");
words.sort(String::compareTo);
```

来考虑一个更复杂的情况，如果有个 List 中某些索引处包括 null，现在你打算让那些 null 排在最前头，之后根据字符串的长度由大到小排序，那会怎么写？这样写吗？

```
public class StrLengthInverseNullFirstComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        if(s1 == s2) {
            return 0;
        }
        if(s1 == null) {
            return -1;
        }
        if(s2 == null) {
            return 1;
        }
```

```
            if(s1.length() == s2.length()) {
                return 0;
            }
            if(s1.length() > s2.length()) {
                return -1;
            }

            return 1;
        }
    }
```

不怎么好读，对吧！更别说为了表示这个比较器的目的，必须取个又臭又长的类别名称！其实排序会有各式各样的组合需求，JDK8考虑到这点，为排序加入了一些高级语义API，例如Comparator上新增了一些静态方法，结合这些方法，可以让程序代码写得具有较高的可读性。

以刚才的需求为例，在JDK8之后要建立对应的Comparator实例，可以如下撰写：

Collection Sort4.java

```
package cc.openhome;

import java.util.*;
import static java.util.Comparator.*;

public class Sort6 {
    public static void main(String[] args) {
        List words = Arrays.asList("B", "X", "A", "M", null ,"F", "W", "O", null);
        words.sort(nullsFirst(reverseOrder()));
        System.out.println(words);
    }
}
```

reverseOrder()返回的Comparator会是Comparable对象上定义顺序的反序，nullsFirst()接受Comparator，在其定义的顺序上加上让null排在最前面的规则后，返回新的Comparator。程序的执行结果如下：

```
[null, null, X, W, O, M, F, B, A]
```

在这里可以看到import static适当的运用，可以让程序码表达出本身操作的意图，相比以下程序代码来说，应该是清楚许多：

```
words.sort(Comparator.nullsFirst(Comparator.reverseOrder()));
```

提示>>> Comparator上还有很多方法可以使用，例如comparing()与thenComparing()等方法，只不过要运用这些方法，你得了解更多JDK8的Lambda特性，例如位于java.util.function套件中的Function等接口的意义，这会在第12章详细介绍。

9.2 键值对应的Map

就如同网络搜索，根据关键字可找到对应的数据，程序设计中也常有这类需求，根据某个

键(Key)来取得对应的值(Value)。可以事先利用 **java.util.Map** 接口的操作对象来建立键值对应数据，之后若要取得值，只要用对应的键就可以迅速取得。

9.2.1 常用 Map 操作类

同样地，在使用 Map 相关 API 前，先了解 Map 设计架构，对正确使用 API 会有帮助，如图 9.11 所示。

常用的 Map 操作类为 **java.util.HashMap** 与 **java.util.TreeMap**，其继承自抽象类 java.util.AbstractMap。至于 java.util.Dictionary 与 java.util.HashTable 是从 JDK1.0 就遗留下来的 API，不建议使用。HashTable 从 JDK1.2 开始操作了 Map 接口，以与 Map API 结合，虽然不建议再使用 HashTable，但子类 **java.util.Properties** 还经常使用，因此这里一并介绍。

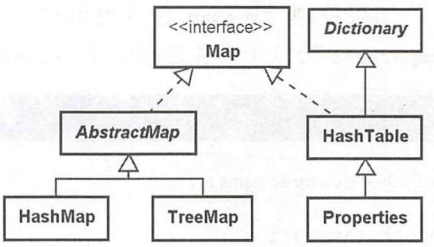


图 9.11 Map 接口操作与继承架构

1. 使用 HashMap

Map 也支持泛型语法，使用上很简单。来看个使用 HashMap 的范例，可以根据指定的用户名称取得对应的信息：

Map Messages.java

```
package cc.openhome;

import java.util.*;
import static java.lang.System.out;

public class Messages {
    public static void main(String[] args) {
        Map<String, String> messages = new HashMap<>();
        messages.put("Justin", "Hello! Justin 的信息！");
        messages.put("Monica", "给 Monica 的悄悄话！");
        messages.put("Irene", "Irene 的可爱猫喵喵叫！");

        Scanner console = new Scanner(System.in);
        out.print("取得谁的信息：");
        String message = messages.get(console.nextLine());
        out.println(message);
        out.println(messages);
    }
}
```

❶ 以泛型语法指定键值类型

❷ 建立键值对应

❸ 指定键返回值

建立 Map 操作对象时，可以使用泛型语法指定键与值的类型。在这里键使用 String，值也使用 String 类型❶。要建立键值对应，可以使用 **put()** 方法，第一个自变量是键，第二个自变量是值❷。对于 Map 而言，键不会重复，判断键是否重复是根据 **hashCode()** 与 **equals()**，所以作为键的对象必须操作 **hashCode()** 与 **equals()**。若要指定键取回对应的值，则使用 **get()** 方法❸。执行结果如下：

严禁网络传播本PDF，违者责任自负！

```
取得谁的信息：Monica
给 Monica 的悄悄话！
{Monica=给 Monica 的悄悄话！, Justin=Hello! Justin 的信息！, Irene=Irene 的可爱猫喵喵叫！}
```

在 HashMap 中建立键值对应之后，键是无序的，这可以在执行结果中看到。如果想让键是有序的，则可以使用 TreeMap。

2. 使用 TreeMap

如果使用 **TreeMap** 建立键值对应，则键的部分将会排序，条件是作为键的对象必须操作 **Comparable** 接口，或者是在创建 **TreeMap** 时指定操作 **Comparator** 接口的对象。例如：

Map Messages2.java

```
package cc.openhome;

import java.util.*;

public class Messages2 {
    public static void main(String[] args) {
        Map<String, String> messages = new TreeMap<>();
        messages.put("Justin", "Hello! Justin 的信息！");
        messages.put("Monica", "给 Monica 的悄悄话！");
        messages.put("Irene", "Irene 的可爱猫喵喵叫！");
        System.out.println(messages);
    }
}
```

由于 String 有操作 Comparable 接口，因此可看到结果会是根据键来排序：

```
{Irene=Irene 的可爱猫喵喵叫！, Justin=Hello! Justin 的信息！, Monica=给 Monica 的悄悄话！}
```

假设想看到相反的排序结果，那么可以这样操作 Comparator：

Map Messages3.java

```
package cc.openhome;

import java.util.*;

public class Messages3 {
    public static void main(String[] args) {
        Map<String, String> messages =
                new TreeMap<>((s1, s2) -> -s1.compareTo(s2));

        messages.put("Justin", "Hello! Justin 的信息！");
        messages.put("Monica", "给 Monica 的悄悄话！");
        messages.put("Irene", "Irene 的可爱猫喵喵叫！");
        System.out.println(messages);
    }
}
```

创建 TreeMap 时指定了 StringComparator 实例，所以执行结果如下：

```
{Monica=给 Monica 的悄悄话！, Justin=Hello! Justin 的信息！, Irene=Irene 的可爱猫喵喵叫！}
```


严禁网络传播本PDF，违者责任自负！

3. 使用 Properties

Properties 类继承自 HashTable，HashTable 操作了 Map 接口，Properties 自然也有 Map 的行为。虽然也可以使用 put() 设定键值对应、get() 方法指定键取回值，不过一般常用 Properties 的 **setProperty()** 指定字符串类型的键值，**getProperty()** 指定字符串类型的键，取回字符串类型的值通常称为属性名称与属性值。例如：

```
Properties props = new Properties();
props.setProperty("username", "justin");
props.setProperty("password", "123456");
out.println(props.getProperty("username"));
out.println(props.getProperty("password"));
```

Properties 也可以从文档中读取属性，例如若有个.properties 文档如下：

Map person.properties

```
# 用户名称与密码
cc.openhome.username=justin
cc.openhome.password=123456
```

.properties 的=左边设定属性名称，右边设定属性值。可以使用 Properties 的 **load()** 方法指定 InputStream 的实例，如 FileInputStream 从文档中加载属性。例如：

Map LoadProperties.java

```
package cc.openhome;

import java.io.*;
import java.util.Properties;

public class LoadProperties {
    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.load(new FileInputStream(args[0]));
        System.out.println(props.getProperty("cc.openhome.username"));
        System.out.println(props.getProperty("cc.openhome.password"));
    }
}
```

load() 方法结束后，会自动关闭 InputStream 实例。就上例而言，如果命令行自变量指定了 person.properties 的位置，则执行结果如下：

```
justin
123456
```

除了可载入.properties 文档外，也可以使用 **loadFromXML()** 方法加载.xml 文档。文件格式必须是：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment></comment>
```


严禁网络传播本PDF，违者责任自负！

```
    <entry key="cc.openhome.username">justin</entry>
    <entry key="cc.openhome.username">123456</entry>
</properties>
```

在使用 java 指令启动 JVM 时，可以使用-D 指定系统属性。例如：

```
> java -Dusername=justin -Dpassword=123456 LoadSystemProps
```

可以使用 System 的 static 方法 **getProperties()** 取得 Properties 实例，该实例包括了系统属性。例如：

Map LoadSystemProps.java

```
package cc.openhome;

import java.util.Properties;

public class LoadSystemProps {
    public static void main(String[] args) {
        Properties props = System.getProperties();
        System.out.println(props.getProperty("username"));
        System.out.println(props.getProperty("password"));
    }
}
```

System.getProperties() 返回的 Properties 实例中，也包括了许多默认属性，例如 java.version 可取得 JRE 版本，java.class.path 可取得类路径等。详细属性可查阅 System.getProperties() 的 API 文件说明，如图 9.12 所示。

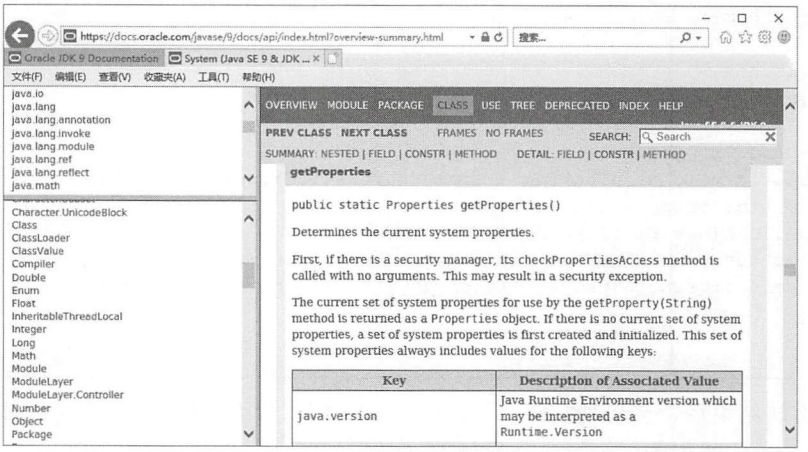


图 9.12　默认的系统属性

9.2.2　访问 Map 键值

如果想取得 Map 中所有的键，或是想取得 Map 中所有的值该怎么做？Map 虽然与 Collection 没有继承上的关系，但它们是彼此搭配的 API。

如果想取得 Map 中所有的键，可以调用 Map 的 **keySet()** 返回 Set 对象。由于键是不重复的，所以用 Set 操作返回是理所当然的做法，如果想取得 Map 中所有的值，则可以使用 **values()** 返回 Collection 对象。例如：

严禁网络传播本PDF，违者责任自负！

Map MapKeyValue.java

```
package cc.openhome;

import java.util.*;
import static java.lang.System.out;

public class MapKeyValue {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("one", "一");
        map.put("two", "二");
        map.put("three", "三");

        out.println("显示键");
        // keySet()返回 Set
        map.keySet().forEach(key -> out.println(key));

        out.println("显示值");
        // values()返回 Collection
        map.values().forEach(key -> out.println(key));
    }
}
```

因为 Set 或 Collection 都是一种 Iterable，所以可使用范例中的 foreach()方法，或者使用如下增强式 for 循环语法。

```
static void foreach(Iterable<String> iterable) {
    for(String element : iterable) {
        out.println(element);
    }
}
```

提示>>> 我们一直强调面向对象基础，以及认识 API 架构的重要性，从这里就可以看出，面对庞大的 API，稳固的面向对象基础以及了解 API 架构，才能够以简驭繁、活用 API。

上面这个范例采用 HashMap 操作，所以执行结果是无序的：

```
显示键
two
one
three
显示值
二
一
三
```

如果将范例改用 TreeMap 操作，则可以看出执行结果将依键排序：

```
显示键
one
three
two
显示值
一
三
二
```


严禁网络传播本PDF，违者责任自负！

如果想同时取得 Map 的键与值，可以使用 **entrySet()**方法，这会返回一个 Set 对象，每个元素都是 **Map.Entry** 实例，可以调用 **getKey()**取得键，调用 **getValue()**取得值。例如：

Map MapKeyValue2.java

```
package cc.openhome;

import java.util.*;

public class MapKeyValue2 {
    public static void main(String[] args) {
        Map<String, String> map = new TreeMap<>();
        map.put("one", "一");
        map.put("two", "二");
        map.put("three", "三");
        foreach(map.entrySet());
    }

    static void foreach(Iterable<Map.Entry<String, String>> iterable) {
        for(Map.Entry<String, String> entry: iterable) {
            System.out.printf("(键 %s, 值 %s)%n",entry.getKey(), entry.getValue());
        }
    }
}
```

这个范例采用了一些较精简的语法，不过初学者可能看不太懂泛型语法的一部分，泛型语法用到某个程度时，可读性并不好，撰写程序还是得兼顾可读性。以下范例执行结果相同，但更容易懂些：

Map MapKeyValue3.java

```
package cc.openhome;

import java.util.*;

public class MapKeyValue3 {
    public static void main(String[] args) {
       Map<String, String> map = new TreeMap<>();
        map.put("one", "一");
        map.put("two", "二");
        map.put("three", "三");
        map.forEach(
           (key, value) -> System.out.printf("(键 %s, 值 %s)%n", key, value)
        );
    }
}
```

Map 没有继承 Iterable，在范例中看到的 forEach()方法是定义在 Map 接口上的，其接受 java.util. function.BiConsumer<T, U>接口实例，这个接口上只有一个抽象方法 void accept(T t, U u)必须操作，两个参数分别接受每次迭代 Map 而得到键与值，结合 Lambda 表达式可获得不错的可读性。两个范例的执行结果都相同，如下所示：

```
(键 one, 值 一)
(键 three, 值 三)
(键 two, 值 二)
```


严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示 »» 有些人喜欢泛型，因为多了编译时期检查；有些人不爱用泛型，因为有时写来太魔幻。无论你是前者还是后者，都应该知道执行时期你使用的对象是哪种类型。

提示 »» 除了 `forEach()` 之外，`Map` 上还有一些好用的方法，可以进一步查看文章《Map 便利的预设方法》：
http:// openhome.cc/Gossip/CodeData/JDK8/Map.html

9.3 不可变的 Collection 与 Map

Java 是个以面向对象为主的经典语言，然而，面向对象并非表现在撰写程序时。近几年来，函数式编程(Functional programming)越来越受到重视，Java 也吸纳了函数式编程的一些概念，前面的 Lambda 表达式就是其中之一。在这一节中，会来看看不可变的(Immutable)`Collection` 与 `Map`。实际上，不可变(Immutability)也是函数式编程中的概念之一。

9.3.1 浅谈不可变特性

不可变(Immutability)是函数式编程的基本特性之一，若你试着去了解函数式编程，会看到有不少说法是这么描述的："纯函数式语言中的变量是不可变的。"是这样的吗？基本上没错，在纯函数式语言中(像是 Haskell)，当你说 `x = 1`，那么无法再修改它的值，`x` 就是 `1`，`1` 的名称(而不是变量)就是 `x`，不会再是其他东西。实际上，在纯函数式语言中，并没有变量的概念存在。

Java 并不是以函数式为主，一开始没有不可变的概念与特性。在 Java 中想使用变量来模仿不可变特性，通常会给变量加上 `final` 修饰，然后试图在这样的限制之上，将程序目的实现出来。

谈到不可变特性，就会相对应地谈到副作用(Side effect)。一个具有副作用的方法会改变对象状态，例如 `Collections` 的 `sort()` 方法，会调整 `List` 实例的元素顺序，这就使得 `List` 实例的状态改变，因而 `sort()` 方法是有副作用的方法。而 `List` 本身的 `add()` 方法，会增加内含元素的数量，这就使得 `List` 实例的状态改变，因而 `List` 本身的 `add()` 方法是有副作用的方法。

一个 Java 应用程序在运行期间，系统本身的状态是在不断变化的，而系统的状态就是由许多对象的状态来组成。如果程序语言本身有变数的概念，在撰写时就容易调整变数值，从而调整对象状态，也就容易变更整个系统状态。

然而，副作用是个双面刃，在一个设计不良的系统中，若没有适当地管控副作用，追踪对象的状态就会越来越困难，最后系统的状态就会难以掌握。常见的问题是除错困难，难以追查系统发生错误的原因。更可怕的是，你明明知道程序应该是写错了，系统却能正常运作，只能担忧着哪天踩到里面的地雷而爆出系统漏洞。

如果变量不可变，那设计出来的方法就不会有副作用，对象状态也会是不可变。不可变对象(Immutable object)有许多好处，例如在并行(Concurrent)程序设计时，就不用担心那些线程共享竞争的问题；在面对数据处理问题若需要一些 `Collection` 对象，例如有序的 `List`、收集不重复对象的 `Set` 等，如果这些对象不可变，那么就有可能共享数据结构，达到节省时间及空间之目的。

Java 毕竟也不是以函数式为主，在设计 `Collection` 框架时，并没有为不可变对象设计专用类型，看看 `Collection` 接口就知道了，那些 `add()`、`remove()` 等方法就直接定义在上面。在

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Collections Framework Overview 中介绍过，有些方法操作都是选用的(Optional)，如果不打算提供操作的方法，可以抛出 **UnsupportedOperationException**，而操作对象必须在文件上指明，支持哪些操作。

提示 »» Collections Framework Overview 的链接：
docs.oracle.com/javase/9/docs/api/java/util/doc-files/coll-overview.html

虽然 Java 不是以函数式为主，但有时在设计上为了限制副作用的发生，会希望某些对象具有不可变的特性，以便掌握对象状态，从而使得系统的某个部分掌握其状态。

由于 Collections 与 Map 在程序中会用来收集与管理对象，为了容易掌握这些对象的状态，在 JDK8 以及之前的版本上，会通过 Collections 上提供的 unmodifiableCollection()、unmodifiableList()、unmodifiableSet()、unmodifiableMap()等 static 方法来取得一个无法修改的(unmodifiable)对象。然而这还不够，为了更进一步支持不可变特性，JDK9 在 List、Set、Map 上直接提供了 of()方法，用以建立不可变的 List、Set 或 Map 操作对象。

实际上，两者都应该认识，而认识 Collections 的 unmodifiableXXX()方法，也会比较清楚了解到，为何 JDK9 在 List、Set、Map 上要提供 of()方法以建立不可变的 List、Set 或 Map 操作对象。

提示 »» 在 JDK8 或旧版 JDK 上，可以通过 guava-libraries 来取得不可变的 List、Set 或 Map 操作对象，你可以先参考《Guava 教学》中的介绍，链接地址如下：
openhome.cc/Gossip/CodeData/GuavaTutorial/

9.3.2 Collections 的 unmodifiableXXX()方法

如果有个 Collection 或 Map 已收集了一些元素，现在打算传送这个对象，而且不希望拿到此对象的任何一方对它做出修改(Modify)操作，那么可以使用 Collections 提供的 unmodifiableXXX()方法，那些方法会返回一个不可修改的对象，单纯为了取得元素是没问题，如果调用了有副作用的 add()、remove()等方法，则会丢出 UnsupportedOperationException，例如：

```
jshell> List<String> names = new ArrayList<>();
names ==> []

jshell> names.add("Monica");
$2 ==> true

jshell> names.add("Justin");
$3 ==> true

jshell> List<String> unmodifiableNames = Collections.unmodifiableList(names);
unmodifiableNames ==> [Monica, Justin]

jshell> unmodifiableNames.get(0);
$5 ==> "Monica"

jshell> unmodifiableNames.add("Irene")
|  java.lang.UnsupportedOperationException thrown:
|        at Collections$UnmodifiableCollection.add (Collections.java:1056)
```

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
|        at (#6:1)
```

那么，通过 unmodifiableXXX()方法返回的对象是不可变吗？不是，返回的对象只是无法修改(Unmodifiable)，也就是仅仅不支持修改操作罢了。这是什么意思？以上面的程序片段来说，如果直接调用 names.add("Irene")，unmodifiableNames 的内容也就跟着变动了：

```
jshell> names.add("Irene");
$7 ==> true

jshell> unmodifiableNames;
unmodifiableNames ==> [Monica, Justin, Irene]
```

之所以会这样，是因为 get()、containsAll()这类方法，只是单纯委托给 unmodifiableXXX()接收它的对象(而 add()等方法直接撰写 throw new UnsupportedOperationException())，例如 unmodifiableCollection()方法的操作是这样的：

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c) {
    return new UnmodifiableCollection<>(c);
}

static class UnmodifiableCollection<E> implements Collection<E>, Serializable {
    private static final long serialVersionUID = 1820017752578914078L;

    final Collection<? extends E> c;

    UnmodifiableCollection(Collection<? extends E> c) {
        if (c==null)
            throw new NullPointerException();
        this.c = c;
    }
    ...
    public boolean add(E e) {
        throw new UnsupportedOperationException();
    }
    public boolean remove(Object o) {
        throw new UnsupportedOperationException();
    }

    public boolean containsAll(Collection<?> coll) {
        return c.containsAll(coll);
    }
    ...
}
```

不可变也从来没在 Collections 那些 unmodifiableXXX()方法的保证中，毕竟名称上也指出了，返回的对象是不可修改，而不是不可变动。无论这是不是在玩文字游戏，如果你要的是更进一步的不可变特性，那么使用 Collections 的 unmodifiableXXX()返回的对象，显然是有所不足。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

9.3.3 List、Set、Map 的 of()方法

JDK9 在 List、Set、Map 等都提供了 of()方法，表面上看来，它们似乎只是建立 List、Set、Map 实例的便捷方法，例如：

```
jshell> List<String> nameLt = List.of("Justin", "Monica");
nameLt ==> [Justin, Monica]

jshell> Set<String> nameSet = Set.of("Justin", "Monica");
nameSet ==> [Monica, Justin]

jshell> Map<String, Integer> scoreMap = Map.of("Justin", 95, "Monica", 100);
scoreMap ==> {Justin=95, Monica=100}
```

比较特别的是 Map.of()，它是采取 Map.of(K1, V1, K2, V2)的方式建立，也就是以键、值、键、值的方式来指定。**List、Set、Map 的 of()方法建立的是不可变对象，你不能对它们调用有副作用的方法，否则会抛出 UnsupportedOperationException，例如：**

```
jshell> nameLt.add("Irene");
|  java.lang.UnsupportedOperationException thrown:
|        at ImmutableCollections.uoe (ImmutableCollections.java:70)
|      at ImmutableCollections$AbstractImmutableList.add (ImmutableCollections.java:76)
|        at (#5:1)

jshell> nameSet.add("Irene");
|  java.lang.UnsupportedOperationException thrown:
|        at ImmutableCollections.uoe (ImmutableCollections.java:70)
|     at ImmutableCollections$AbstractImmutableSet.add (ImmutableCollections.java:280)
|        at (#6:1)

jshell> scoreMap.put("Irene", 100);
|  java.lang.UnsupportedOperationException thrown:
|        at ImmutableCollections.uoe (ImmutableCollections.java:70)
|     at ImmutableCollections$AbstractImmutableMap.put (ImmutableCollections.java:557)
|        at (#7:1)
```

那么可以避免刚才 Collections 的 unmodifiableXXX()上提到的问题吗？这些 of()方法多数都是采用可变长度自变量的方式定义，而且重载了多个不同参数个数的版本，以 List 的 of()方法为例，如图 9.13 所示。

在自变量少于 10 个的情况下，会使用对应个数的 of()版本，因而不会有参考原 List 实例的问题。至于那个 of(E… elements)版本，内部并不会直接参考原本 elements 参考的实例，而是建立一个新数组，然后对 elements 的元素逐一浅层复制。下面列出 JDK 中的原始码实作片段以便了解：

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

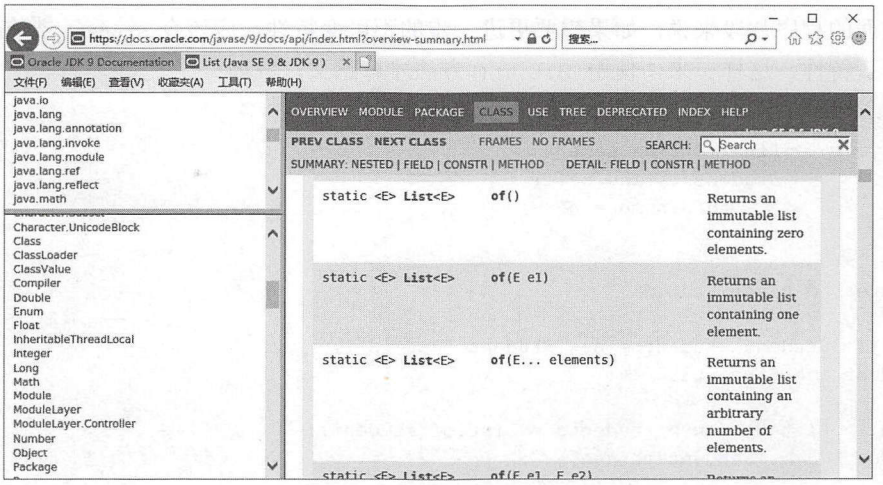


图 9.13 List 的 of()方法

```
ListN(E... input) {
    // copy and check manually to avoid TOCTOU
    @SuppressWarnings("unchecked")
    E[] tmp = (E[])new Object[input.length]; // implicit nullcheck of input
    for (int i = 0; i < input.length; i++) {
        tmp[i] = Objects.requireNonNull(input[i]);
    }
    this.elements = tmp;
}
```

因此在数据结构上，就算对该版本的 of()方法直接传入数组，也没有参考至原 elements 参考对象的疑虑，从而更进一步支持了不可变特性。然而要注意，因为元素是浅层复制，如果直接变更了元素的状态，of()方法返回的对象还是会反映出对应的状态变更。例如：

```
jshell> class Student {
   ...>     String name;
   ...> }
|  created class Student

jshell> Student student = new Student();
student ==> Student@cb644e

jshell> student.name = "Justin";
$3 ==> "Justin"

jshell> List<Student> students = List.of(student);
students ==> [Student@cb644e]

jshell> students.get(0).name;
$5 ==> "Justin"

jshell> student.name = "Monica";
$6 ==> "Monica"

jshell> students.get(0).name;
$7 ==> "Monica"
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

以上面的程序片段来说，如果想要更进一步的不可变特性，应该令 Student 类在定义时也支持不可变特性，这样一来，使用 List.of() 方法才有意义，例如：

```
jshell> class Student {
   ...>     final String name;
   ...>     Student(String name) {
   ...>         this.name = name;
   ...>     }
   ...> }
|  created class Student

jshell> Student student = new Student("Justin");
student ==> Student@cb644e

jshell> List<Student> students = List.of(student);
students ==> [Student@cb644e]
```

你也许会想到前面使用过的 Arrays.asList() 方法，似乎与 List.of() 方法很像。Arrays.asList() 方法返回的对象长度固定，确实也是无法修改。由于方法定义时使用不定长度自变量，也可以直接指定数组作为自变量，这就会引发类似的问题：

```
jshell> String[] names = {"Justin", "Monica"};
names ==> String[2] { "Justin", "Monica" }

jshell> List<String> nameLt = Arrays.asList(names);
nameLt ==> [Justin, Monica]

jshell> names[0] = "Irene";
$3 ==> "Irene"

jshell> nameLt;
nameLt ==> [Irene, Monica]
```

会发生这个问题的理由类似，Arrays.asList() 返回的对象，内部参考了 names 参考之对象(你可以试着查看 Arrays.java 的原始码操作来验证)；如果你需要的是不可变对象，而不是无法修改的对象，那么在 JDK9 之后，建议改用 List.of()，而不是 Arrays.asList()。

9.4 重点复习

收集对象的行为，如新增对象的 add() 方法、移除对象的 remove() 方法等，都是定义在 java.util.Collection 中。既然可以收集对象，也能逐一取得对象，这就是 java.lang.Iterable 定义的行为，它定义了 iterator() 方法返回 java.util.Iterator 操作对象，可以让你逐一取得收集的对象。

如果希望收集时记录每个对象的索引顺序，并可依索引取回对象，这样的行为定义在 java.util.List 接口中。如果希望收集的对象不重复，具有集合的行为，则由 java.util.Set 定义。如果希望收集对象时以队列方式，收集的对象加入至尾端，取得对象时从前端，则可以使用 java.util.Queue。如果希望对 Queue 的两端进行加入、移除等操作，则可以使用 java.util.Deque。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

数组在内存中会是连续的线性空间，根据索引随机存取时速度快，如果操作上有这类需求时，如排序，就可使用 ArrayList，可得到较好的速度表现。

数组在内存中会是连续的线性空间，如果需要调整索引顺序时，表现会较差。数组的长度固定也是要考虑的问题，在 ArrayList 内部数组长度不够时，会建立新数组，并将旧数组的参考指定给新数组，这也是必须耗费时间与内存的操作，ArrayList 有个可指定容量的构造函数。如果大致知道将收集的对象范围，事先建立足够长度的内部数组，可以节省以上所描述的成本。

LinkedList 在操作 List 接口时，采用了链接结构，不会事先耗费内存，在指定索引随机存取对象时，效率较低，链接的每个元素都会参考下一个元素，这有利于调整索引顺序。若收集的对象经常会有变动索引的情况，也许考虑链接方式操作的 List 会比较好。

Java 中要判断对象是否重复时，会调用 hashCode()与 equals()方法，因此建议，两个方法必须同时操作。

如果对象操作 Queue，并打算以队列方式使用，且队列长度受限，通常建议使用 offer()、poll()与 peek()等方法。想对队列的前端与尾端进行操作，在前端加入对象与取出对象，在尾端加入对象与取出对象，Queue 的子接口 Deque 就定义了这类行为。

无论 List、Set 还是 Queue，都会有个 iterator()方法，这个方法在 JDK1.4 之前定义在 Collection 接口中，而 List、Set、Queue 继承自 Collection，所以也都拥有 iterator()的行为。在 JDK5 之后，原先定义在 Collection 中的 iterator()方法，提升至新的 java.util.Iterable 父接口，增强式 for 循环可运用在数组上，还可运用在操作 Iterable 接口的对象上。增强式 for 循环是编译程序蜜糖。

在 Java 的规范中，跟顺序有关的行为，通常要不对象本身是 Comparable，要不就是另行指定 Comparator 对象告知如何排序。

从 JDK5 之后，新增了泛型语法，在设计 API 时可以指定类或方法支持泛型，而使用 API 的客户端在语法上会更为简洁，并得到编译时期检查。

接口只有一个方法必须操作时，在 JDK8 中可以使用 Lambda 表达式取代匿名类语法。

如果使用 JDK8，想要迭代对象还有新的选择，Iterable 上新增了 forEach()方法，可以让迭代对象进行指定处理。

在 Java 的规范中，跟顺序有关的行为，通常要不对象本身是 Comparable，要不就是另行指定 Comparator 对象告知如何排序。JDK8 在 List 上增加了 sort()方法，可接受 Comparator 实例来指定排序方式 JDK8 为排序加入了一些高级语义 API，例如 Comparator 上新增了一些静态方法，结合这些方法可以让程序代码具有较高的可读性。

可以事先利用 java.util.Map 接口的操作对象来建立键值对应数据，之后若要取得值，只要用对应的键就可以迅速取得。判断键是否重复根据 hashCode()与 equals()，所以作为键的对象必须操作 hashCode()与 equals()。

一般常用 Properties 的 setProperty()指定字符串类型的键值，getProperty()指定字符串类型的键，取回字符串类型的值通常称为属性名称与属性值。

如果想取得 Map 中所有键，可以调用 Map 的 keySet()返回 Set 对象；如果想取得 Map 中所有的值，则可以使用 values()返回 Collection 对象。如果想同时取得 Map 的键与值，可以使用 entrySet()方法，这会返回一个 Set 对象，每个元素都是 Map.Entry 实例，可以调用 getKey()取得键，调用 getValue()取得值。

Map 没有继承 Iterable，有个 forEach()方法是定义在 Map 接口上，可使用这个方法结合

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Lambda表达式，在迭代键与值时获得不错的可读性。

通过unmodifiableXXX()方法返回的的对象只是无法修改，也就是仅仅不支持修改操作，如果调用了有副作用的add()、remove()等方法，会抛出UnsupportedOperationException，然后若变动传给unmodifiableXXX()方法的对象，unmodifiableXXX()方法返回的对象状态也会变动。

JDK9新增的List、Set、Map的of()方法建立的是不可变对象，不能对它们调用有副作用的方法，否则会抛出UnsupportedOperationException。

9.5 课后练习

1. 尝试写个IterableString类，可指定字符串创建IterableString实例，让该实例可使用增强式for循环，或者是本身的forEach()方法，逐一取出字符串中的字符。

2. 如果有个字符串数组如下：

```
String[] words = {"RADAR","WARTER START","MILK KLIM","RESERVERED","IWI"};
```

请撰写程序，判断字符串数组中有哪些字符串，从前面看的字符顺序与从后面看的字符顺序是相同的。

提示>>> 可使用Deque。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 10

输入/输出

学习目标

- 了解串流与输入/输出的关系
- 认识 InputStream、OutputStream 继承架构
- 认识 Reader、Writer 继承架构
- 使用输入/输出装饰器类

10.1 InputStream 与 OutputStream

想活用输入/输出 API，一定要先了解 Java 中如何以串流(Stream)抽象化输入/输出概念，以及 InputStream、OutputStream 继承架构。如此一来，无论标准输入/输出、文档输入/输出、网络输入/输出、数据库输入/输出等都可用一致的操作进行处理。

10.1.1 串流设计的概念

Java 将输入/输出抽象化为串流，数据有来源及目的地，衔接两者的是串流对象。比喻来说，数据就好比水，串流好比水管，通过水管的衔接，水由一端流向另一端，如图 10.1 所示。

从应用程序角度来看，如果要将数据从来源取出，可以使用输入串流，如果要将数据写入目的地，可以使用输出串流。在 Java 中，输入串流代表对象为 **java.io.InputStream** 实例，输出串流代表对象为 **java.io.OutputStream** 实例。无论数据源或目的地为何，只要设法取得 InputStream 或 OutputStream 的实例，接下来操作输入/输出的方式都是一致的，无须理会来源或目的地的真正形式，如图 10.2 所示。

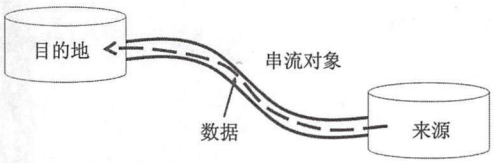


图 10.1 串流衔接来源与目的地

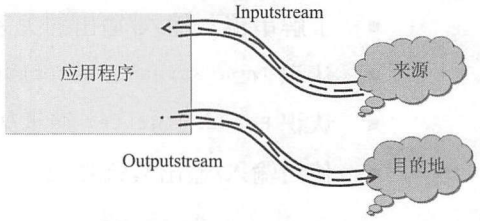


图 10.2 从应用程序看 InputStream 与 OutputStream

在来源与目的地都不知道的情况下，如何撰写程序？听来不可思议，但实际上就是会有这类需求。举个例子，设计一个通用的 dump() 方法：

Stream IO.java

```
package cc.openhome;

import java.io.*;

public class IO {
    public static void dump(InputStream src, OutputStream dest) ←— ❶ 数据来源与目的地
                                throws IOException { ←— ❷ 客户端要处理异常
        try (InputStream input = src; OutputStream output = dest) { ←— ❸ 尝试自动关闭资源
            byte[] data = new byte[1024]; ←— ❹ 尝试每次从来源读取 1024 字节
            int length;
            while ((length = input.read(data)) != -1) { ←— ❺读取数据
                output.write(data, 0, length); ←— ❻ 写出数据
            }
        }
    }
}
```

dump() 方法接受 InputStream 与 OutputStream 实例，分别代表读取数据的来源以及输出数

据的目的地❶。在进行 InputStream 与 OutputStream 的相关操作时若发生错误，会抛出 java.io.IOException 异常，在这里不特别处理，而是在 dump() 方法上声明 throws，由调用 dump() 方法的客户端处理❷。

提示 »» JDK 中许多 I/O 操作会声明抛出 IOException，如果想考虑捕捉后转为执行时期异常，可以使用 JDK8 新增的 java.io.UncheckedIOException，这个异常继承自 RuntimeException。

在不使用 InputStream 与 OutputStream 时，必须使用 **close()** 方法关闭串流。由于 InputStream 与 OutputStream 操作了 **java.io.Closeable** 接口，其父接口为 **java.lang.AutoCloseable** 接口，因此可使用 JDK7 尝试自动关闭资源语法❸。

提示 »» 思考一下，如果不能使用 JDK7 尝试自动关闭资源语法，那使用 try、catch、finally 该怎么写？可以参考一下 8.2.2 节的内容。

每次从 InputStream 读入的数据，都会先置入 byte 数组中❹。InputStream 的 **read()** 方法，每次会尝试读入 byte 数组长度的数据，并返回实际读入的字节，只要不是-1，就表示读取到数据❺。可以使用 OutputStream 的 **write()** 方法，指定要写出的 byte 数组、初始索引与数据长度❻。

那么这个 dump() 方法的来源是什么？不知道。目的地呢？也不知道。dump() 方法并没有限定来源或目的地的真实形式，而是依赖于抽象的 InputStream、OutputStream。如果要将某个文档读入并另存为另一个文档，则可以这么使用：

Stream Copy.java

```
package cc.openhome;

import java.io.*;

public class Copy {
    public static void main(String[] args) throws IOException {
        IO.dump(
              new FileInputStream(args[0]),
              new FileOutputStream(args[1])
        );
    }
}
```

这个程序可以由命令行自变量指定读取的文档来源与写出的目的地，例如：

```
> java cc.openhome.Copy c:\workspace\Main.java C:\workspace\Main.txt
```

稍后就会介绍串流继承架构，FileInputStream 是 InputStream 的子类，用于衔接文档以读入数据；FileOutputStream 是 OutputStream 的子类，用于衔接文档以写出数据。

如果要从 HTTP 服务器读取某个网页，并另存为文档，也可以使用这里设计的 dump() 方法。例如：

Stream Download.java

```
package cc.openhome;

import java.io.*;
import java.net.URL;
```

```
public class Download {
    public static void main(String[] args) throws IOException {
        URL url = new URL(args[0]);
        InputStream src = url.openStream();
        OutputStream dest = new FileOutputStream(args[1]);
        IO.dump(src, dest);
    }
}
```

虽然没有正式介绍到网络程序设计，不过 java.net.URL 的使用很简单，只要指定网址，URL 实例会自动进行 HTTP 协议。可以使用 openStream()方法取得 InputStream 实例，代表与网站连接的数据串流。

无论来源或目的地实体形式为何，只要想办法取得 InputStream 或 OutputStream，接下来都是调用 InputStream 或 OutputStream 的相关方法。例如，使用 java.net.ServerSocket 接受客户端联机的例子：

```
ServerSocket server = null;
Socket client = null;
try {
    server = new ServerSocket(port);
    while(true) {
        client = server.accept();
        InputStream input = client.getInputStream();
        OutputStream output = client.getOutputStream();
        // 接下来就是操作 InputStream、OutputStream 实例了
        ...
    }
}
catch(IOException ex) {
    ...
}
```

如果将来学到 Servlet，想把文档输出至浏览器，也会有类似的操作：

```
response.setContentType("application/pdf");
InputStream in = this.getServletContext()
                     .getResourceAsStream("/WEB-INF/jdbc.pdf");
OutputStream out = response.getOutputStream();
byte[] data = new byte[1024];
int length;
while((length = in.read(data)) != -1) {
    out.write(data, 0, length);
}
```

10.1.2 串流继承架构

在了解串流抽象化数据源与目的地的概念后，接下来要搞清楚 Java 中 InputStream、OutputStream 的继承架构。首先看到 InputStream 的常用类继承架构，如图 10.3 所示。

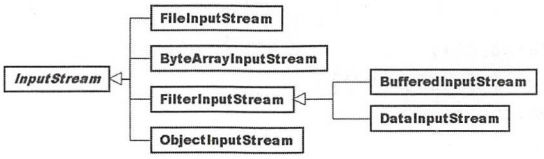


图 10.3 InputStream 常用类继承架构

再来看 OutputStream 的常用类继承架构，如图 10.4 所示。

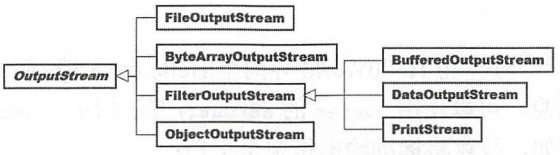


图 10.4 OutputStream 常用类继承架构

了解 InputStream 与 OutputStream 类继承架构之后，再来逐步说明相关类的使用方式。

1. 标准输入/输出

还记得 **System.in** 与 **System.out** 吗？查看 API 文件，会发现它们分别是 **InputStream** 与 **PrintStream** 的实例，分别代表标准输入(Standard Input)与标准输出(Standard Output)，对个人计算机而言，通常对应至文本模式中的输入与输出。

以 System.in 而言，因为文本模式下通常是取得整行用户输入，因此较少直接操作 InputStream 相关方法，而是如前面章节使用 java.util.Scanner 打包 System.in，你操作 Scanner 相关方法，而 Scanner 会代你操控 System.in 取得数据，并转换为取得你想要的数据类型。

可以使用 System 的 **setIn()** 方法指定 InputStream 实例，重新指定标准输入来源。例如下面范例故意将标准输入指定为 FileInputStream，以读取指定文档并显示在文本模式：

Stream StandardIn.java

```
package cc.openhome;

import java.io.*;
import java.util.*;

public class StandardIn {
    public static void main(String[] args) throws IOException {
        System.setIn(new FileInputStream(args[0]));
        try (Scanner file = new Scanner(System.in)) {
            while (file.hasNextLine()) {
                System.out.println(file.nextLine());
            }
        }
    }
}
```

System.out 为 PrintStream 实例，从图 10.4 来看，它是一种 OutputStream，所以若要将 10.1.1 节的 Download 范例改为输出至标准输出，也可以这么写：

```
...
URL url = new URL(args[0]);
InputStream src = url.openStream();
IO.dump(src, System.out);
...
```

标准输出可以重新导向至文档，只要执行程序时使用>将输出结果导向至指定的文档。例如，若 Hello 类执行了 System.out.println("HelloWorld")：

```
> java Hello > Hello.txt
```

那么上面的指令执行方式，将 HelloWorld 导向至 Hello.txt 文档，而不会显示在文本模式中，如果使用>>则是附加信息。可以使用 System 的 **setOut()** 方法指定 PrintStream 实例，将结果输出至指定的目的地。例如，故意将标准输出指定至文档：

Stream StandardOut.java

```
package cc.openhome;

import java.io.*;

public class StandardOut {
    public static void main(String[] args) throws IOException {
        try (PrintStream printStream = new PrintStream(
                new FileOutputStream(args[0]))) {
            System.setOut(printStream);
            System.out.println("HelloWorld");
        }
    }
}
```

PrintStream 接受 InputStream 实例，在这个范例中用 PrintStream 打包 FileOutputStream，你操作 PrintStream 相关方法，PrintStream 会代你操作 FileOutputStream。

除了 System.in 与 System.out 之外，还有个 **System.err** 为 PrintSteam 实例，称为标准错误输出串流，它是用来立即显示错误信息。例如，在文本模式下，System.out 输出的信息可以使用>或>>重新导向至文档，但 System.err 输出的信息一定会显示在文本模式中，无法重新导向。也可以使用 **System.setErr()** 指定 PrintStream，重新指定标准错误输出串流。

2. FileInputStream 与 FileOutputStream

FileInputStream 是 InputStream 的子类，可以指定文件名创建实例，一旦创建文档就开启，接着就可用来读取数据。FileOutputStream 是 OutputStream 的子类，可以指定文件名创建实例，一旦创建文档就开启，接着就可以用来写出数据。无论 FileInputStream 还是 FileOutputStream，不使用时都要用 **close()** 关闭文档。

FileInputStream 主要操作了 InputStream 的 read() 抽象方法，使之可从文档中读取数据。FileOutputStream 主要操作了 OutputStream 的 write() 抽象方法，使之可写出数据至文档。前面的 IO.dump() 方法中已示范过 read() 与 write() 方法。

FileInputStream、FileOutputStream 在读取、写入文档时，是以字节为单位，通常会使用一些高阶类加以打包，进行一些高阶操作，像是前面示范过的 Scanner 与 PrintStream 类等。之后还会看到更多打包 InputStream、OutpuStream 的类，它们也可以用来打包 FileInputStream、

FileOutputStream。

3. ByteArrayInputStream 与 ByteArrayOutputStream

ByteArrayInputStream 是 InputStream 的子类，可以指定 byte 数组创建实例，一旦创建就可将 byte 数组当作数据源进行读取。ByteArrayOutputStream 是 OutputStream 的子类，可以指定 byte 数组创建实例，一旦创建将 byte 数组当作目的地写出数据。

ByteArrayInputStream 主要操作了 InputStream 的 read()抽象方法，使之可从 byte 数组中读取数据。ByteArrayOutputStream 主要操作了 OutputStream 的 write()抽象方法，使之可写出数据至 byte 数组。前面的 IO.dump()方法中示范过的 read()与 write()方法，就是 ByteArrayInputStream、ByteArrayOutputStream 的操作范例，毕竟它们都是 InputStream、OutputStream 的子类。

10.1.3 串流处理装饰器

InputStream、OutputStream 提供串流基本操作，如果想要为输入/输出的数据做加工处理，则可以使用打包器类。前面示范过的 Scanner 类就是作为打包器，其接受 InputStream 实例，你操作 Scanner 打包器相关方法，Scanner 会实际操作打包的 InputStream 取得数据，并转换为你想要的数据类型。

InputStream、OutputStream 的一些子类也具有打包器的作用。这些子类在创建时，可以接受 InputStream、OutputStream 实例。前面介绍的 PrintStream 就是实际例子，你操作 PrintStream 的 print()、println()等方法，PrintStream 会自动转换为 byte 数组数据，利用打包的 OutputStream 进行输出。

常用的打包器有具备缓冲区作用的 BufferedInputStream、BufferedOutputStream，具备数据转换处理作用的 DataInputStream、DataOutputStream，具备对象串行化能力的 ObjectInputStream、ObjectOutputStream 等。

由于这些类本身并没有改变 InputStream、OutputStream 的行为，只不过在 InputStream 取得数据之后，再做一些加工处理，或者是要输出时做一些加工处理，再交由 OutputStream 真正进行输出，因此又称它们为装饰器(Decorator)。就像给照片装上华丽外框，可以让照片感觉更为华丽，或有点像小水管衔接大水管，如小水管(InputStream)读入数据，再由大水管(如 BufferedInputStream)增加缓冲功能，如图 10.5 所示。

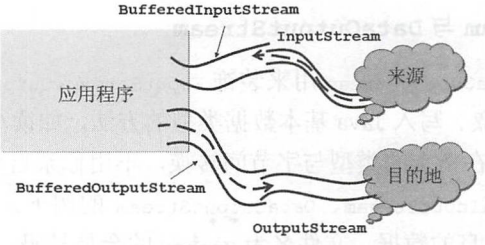


图 10.5 装饰器提供高阶操作

下面介绍几个常用的串流装饰器类。

1. BufferedInputStream 与 BufferedOutputStream

在前面 IO.dump()方法中，每次调用 InputStream 的 read()方法，都会直接向来源要求数据，

每次调用 OutputStream 的 write()方法时，都会直接将数据写到目的地，这并不是个有效率的方式。

以文档存取为例，如果传入 IO.dump()的是 FileInputStream、FileOutputStream 实例，每次 read()时都会要求读取硬盘，每次 write()时都会要求写入硬盘，这会花费许多时间在硬盘定位上。

如果 InputStream 第一次 read()时可以尽量读取足够的数据至内存的缓冲区，后续调用 read()时先看看缓冲区是不是还有数据，如果有就从缓冲区读取，没有再从来源读取数据至缓冲区，这样可以减少从来源直接读取数据的次数，对提高读取效率将会有帮助。

如果 OutputStream 每次 write()时可将数据写入内存中的缓冲区，缓冲区满了再将缓冲区的数据写入目的地，这样可减少对目的地的写入次数，对提高写入效率将会有帮助。

BufferedInputStream 与 BufferedOutputStream 提供的就是前面描述的缓冲区功能，创建 BufferedInputStream、BufferedOutputStream 必须提供 InputStream、OutputStream 进行打包，可以使用默认或自定义缓冲区大小。

BufferedInputStream 与 BufferedOutputStream 主要在内部提供缓冲区功能，操作上与 InputStream、OutputStream 并没有太大差别。例如，改写前面的 IO.dump()为 BufferedIO.dump()方法：

Stream BufferedIO.java

```
package cc.openhome;

import java.io.*;

public class BufferedIO {
    public static void dump(InputStream src, OutputStream dest)
                                throws IOException {
        try(InputStream input = new BufferedInputStream(src);
             OutputStream output = new BufferedOutputStream(dest)) {
            byte[] data = new byte[1024];
            int length;
            while ((length = input.read(data)) != -1) {
                output.write(data, 0, length);
            }
        }
    }
}
```

2. DataInputStream 与 DataOutputStream

DataInputStream、DataOutputStream 用来装饰 InputStream、OutputStream，DataInputStream、DataOutputStream 提供读取、写入 Java 基本数据类型的方法，如读写 int、double、boolean 等的方法。这些方法会自动在指定的类型与字节间转换，不用你亲自做字节与类型转换的动作。

来看个实际使用 DataInputStream、DataOutputStream 的例子。下面的 Member 类可以调用 save()储存 Member 实例本身的数据。文件名为 Member 的会员号码，调用 Member.load()指定会员号码，则可以读取文档中的会员数据，封装为 Member 实例并返回。

Stream Member.java

```
package cc.openhome;

import java.io.*;
```


非卖品，仅供非商业用途或交流学习使用

```
public class Member {
    private String number;
    private String name;
    private int age;

    public Member(String number, String name, int age) {
        this.number = number;
        this.name = name;
        this.age = age;
    }
    // 部分程序代码省略...因为只是一些 Getter、Setter...

    @Override
    public String toString() {
        return String.format("(%s, %s, %d)", number, name, age);
    }
    public void save() throws IOException {
        try(DataOutputStream output =                    ❶ 建立 DataOutputStream 打包 FileOutputStream
                new DataOutputStream(new FileOutputStream(number))) {
            output.writeUTF(number);
            output.writeUTF(name);                       ❷ 根据不同的类型调用 writeXXX()方法
            output.writeInt(age);
        }
    }

    public static Member load(String number) throws IOException {
        Member member;
        try(DataInputStream input =                      ❸ 建立 DataInputStream 打包 FileInputStream
                new DataInputStream(new FileInputStream(number))) {
            member = new Member(
                    input.readUTF(), input.readUTF(), input.readInt());
        }                                                          ❹ 根据不同的类型调用
        return member;                                               readXXX()方法
    }
}
```

在 save()方法中，使用 DataOutputStream 打包 FileOutputStream❶，在储存 Member 实例时，会使用 writeUTF()、writeInt()方法分别储存字符串与 int 类型❷。在 load()方法中，则使用 DataInputStream 打包 FileInputStream❸，并调用 readUTF()、readInt()分别读入字符串、int 类型❹。下面是个使用 Member 类的例子：

Stream MemberDemo.java

```
package cc.openhome;

import java.io.IOException;
import static java.lang.System.out;

public class MemberDemo {
    public static void main(String[] args) throws IOException {
        Member[] members = {
                    new Member("B1234", "Justin", 90),
                    new Member("B5678", "Monica", 95),
                    new Member("B9876", "Irene", 88)
        };
        for(Member member : members) {
            member.save();
        }
```


非卖品，仅供非商业用途或交流学习使用

```
        out.println(Member.load("B1234"));
        out.println(Member.load("B5678"));
        out.println(Member.load("B9876"));
    }
}
```

范例中准备了三个 Member 实例，分别储存为文档之后再读取回来。执行结果如下：

```
(B1234, Justin, 90)
(B5678, Monica, 95)
(B9876, Irene, 88)
```

3. ObjectInputStream 与 ObjectOutputStream

前面的范例是取得 Member 的 number、name、age 数据进行储存，读取时也是先取得 number、name、age 数据再用来创建 Member 实例。实际上，也可以将内存中的对象整个储存下来，之后再读入还原为对象。可以使用 ObjectInputStream、ObjectOutputStream 装饰 InputStream、OutputStream 来完成这项工作。

ObjectInputStream 提供 **readObject()** 方法将数据读入为对象，而 ObjectOutputStream 提供 **writeObject()** 方法将对象写至目的地。可以被这两个方法处理的对象，必须操作 **java.io.Serializable** 接口，这个接口并没有定义任何方法，只是作为标示之用，表示这个对象是可以串行化的(Serializable)。

下面这个范例改写了前一个范例，使用 ObjectInputStream、ObjectOutputStream 来储存、读入数据。

Stream Member2.java

```
package cc.openhome;

import java.io.*;

public class Member2 implements Serializable { ←— ❶ 操作 Serializable
    private String number;
    private String name;
    private int age;

    public Member2(String number, String name, int age) {
        this.number = number;
        this.name = name;
        this.age = age;
    }
    // 部分程序代码省略...因为只是一些 Getter、Setter...

    @Override
    public String toString() {
        return String.format("(%s, %s, %d)", number, name, age);
    }

    public void save() throws IOException {
        try(ObjectOutputStream output = ←— ❷ 建立 Object OutputStream 打包
                new ObjectOutputStream(new FileOutputStream(number))) {
            output.writeObject(this); ←— ❸ 调用 writeObject()方法写出对象
        }
    }
```


非卖品，仅供非商业用途或交流学习使用

```
    public static Member2 load(String number)
                        throws IOException, ClassNotFoundException {
        Member2 member;
        try(ObjectInputStream input = ←——❹建立 Object InputStream 打包 FileInputStream
                new ObjectInputStream(new FileInputStream(number))) {
            member = (Member2) input.readObject(); ←——❺调用 readObject()方法写入对象
        }
        return member;
    }
}
```

为了能够直接将对象写出或读入，Member2 操作了 Serializable❶，在储存对象时，使用 ObjectOutputStream 打包 FileOutputStream❷，ObjectOutputStream 的 writeObject()处理内存中的对象数据，再交给 FileOutputStream 写至文档❸。在读入对象时，使用 ObjectInputStream 打包 FileInputStream❹，在 readObject()时，会用 FileInputStream 读入字节数据，再交给 ObjectInputStream 处理，还原为 Member2 实例❺。

下面的程序用来测试 Member2 类是否可正确写出与读入对象，执行结果与 MemberDemo 是相同的：

Stream Member2Demo.java

```
package cc.openhome;

import static java.lang.System.out;

public class Member2Demo {
    public static void main(String[] args) throws Exception {
        Member2[] members = {new Member2("B1234", "Justin", 90),
                             new Member2("B5678", "Monica", 95),
                             new Member2("B9876", "Irene", 88)};
        for(Member2 member : members) {
            member.save();
        }
        out.println(Member2.load("B1234"));
        out.println(Member2.load("B5678"));
        out.println(Member2.load("B9876"));
    }
}
```

如果在做对象串行化时，对象中某些数据成员不希望被写出，则可以标上 **transient** 关键字。

10.2 字符处理类

InputStream、OutputStream 是用来输入与输出字节数据，若实际上处理的是字符数据，使用 InputStream、OutputStream 就得对照编码表，在字符与字节之间进行转换。所幸 Java SE API 已提供相关输入/输出字符处理类，让你不用亲自进行字节与字符编码转换的枯燥工作。

非卖品，仅供非商业用途或交流学习使用

10.2.1 Reader 与 Writer 继承架构

针对字符数据的读取，Java SE 提供了 **java.io.Reader** 类，其抽象化了字符数据读取的来源。针对字符数据的写入，则提供了 **java.io.Writer** 类，其抽象化了数据写出的目的地。

举个例子来说，如果想从来源读取字符数据，或将字符数据写至目的地，都可以使用下面的 CharUtil.dump()方法：

Stream CharUtil.java

```
package cc.openhome;

import java.io.*;

public class CharUtil {
    public static void dump(Reader src, Writer dest) throws IOException {
        try(Reader input = src; Writer output = dest) {
            char[] data = new char[1024];
            int length;
            while((length = src.read(data)) != -1) {
                dest.write(data, 0, length);
            }
        }
    }
}
```

❶ 数据来源与目的地
❷ 客户端要处理异常
❸ 尝试自动关闭资源
❹ 尝试每次从来源读取 1024 字符
❺ 读取数据
❻ 写出数据

dump()方法接受 Reader 与 Writer 实例，分别代表读取数据的来源以及输出数据的目的地❶。在进行 Reader 与 Writer 的相关操作时若发生错误，会抛出 IOException 异常，在这里不特别处理，而是在 dump()方法上声明 throws，由调用 dump()方法的客户端处理❷。

在不使用 Reader 与 Writer 时，必须使用 **close()**方法关闭串流。由于 Reader 与 Writer 操作了 **Closeable** 接口，其父接口为 **AutoCloseable** 接口，因此可使用 JDK7 尝试自动关闭资源语法❸。

每次从 Reader 读入的数据，都会先置入 char 数组中❹。Reader 的 **read()**方法，每次会尝试读入 char 数组长度的数据，并返回实际读入的字符数，只要不是-1，就表示读取到字符❺。可以使用 Writer 的 **write()**方法，指定要写出的 char 数组、初始索引与数据长度❻。

同样地，了解 Reader、Writer 继承架构有利于 API 的灵活运用。首先看 Reader 继承架构，如图 10.6 所示。

图 10.6 中列出了几个常用的 Reader 子类。再来看看 Writer 常用类继承架构，如图 10.7 所示。

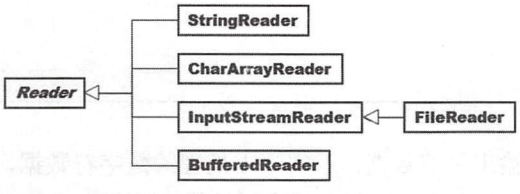


图 10.6 Reader 继承架构

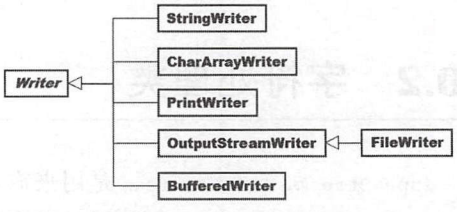


图 10.7 Writer 继承架构

从图 10.6 与图 10.7 得知，FileReader 是一种 Reader，主要用于读取文档并将读到的数据转换为字符；StringWriter 是一种 Writer，可以将字符数据写至 StringWriter，最后使用 toString()方法取得字符串，代表所有写入的字符数据。所以，若要使用 CharUtil.dump()读入文档、转为

非卖品，仅供非商业用途或交流学习使用

字符串并显示在文本模式中。如：

Stream CharUtilDemo.java

```
package cc.openhome;
import java.io.*;
public class CharUtilDemo {
    public static void main(String[] args) throws IOException {
        FileReader reader = new FileReader(args[0]);
        StringWriter writer = new StringWriter();
        CharUtil.dump(reader, writer);
        System.out.println(writer.toString());
    }
}
```

如果执行 CharUtilDemo 时，在命令行自变量指定了文档位置，而文档中实际都是字符数据，就可以在文本模式中看到文档中的文字内容。

稍微解释一下几个常用的 Reader、Writer 子类。StringReader 可以将字符串打包，当作读取来源，StringWriter 则可以作为写入目的地，最后用 toString()取得所有写入的字符组成的字符串。CharArrayReader、CharArrayWriter 则类似，将 char 数组当作读取来源以及写入目的地。

FileReader、FileWriter 可以对文档做读取与写入，读取或写入时默认会使用操作系统默认编码来做字符转换。也就是说，如果你的操作系统默认编码是 GB2312，则 FileReader、FileWriter 会以 GB2312 对你的“纯文本文档”做读取、写入的动作，如果操作系统默认编码是 UTF-8，则 FileReader、FileWriter 就使用 UTF-8。

在启动 JVM 时，可以通过**-Dfile.encoding** 来指定 FileReader、FileWriter 所使用的编码。例如，指定使用 UTF-8：

```
> java -Dfile.encoding=UTF-8 cc.openhome.CharUtil sample.txt
```

FileReader、FileWriter 没有可以指定编码的方法。如果在程序执行过程中想要指定编码，则必须使用 InputStreamReader、OutputStreamWriter，这两个类实际上是作为装饰器。在 10.2.2 节中一并说明。

提示 »» 纯文本文档？编码？如果你看到这些名词不太懂的话，建议参考一下文章《乱码 1/2》：http://openhome.cc/Gossip/Encoding/

10.2.2 字符处理装饰器

如同 InputStream、OutputStream 有一些装饰器类，可以对 InputStream、OutputStream 打包增加额外功能，Reader、Writer 也有一些装饰器类可供使用。下面介绍常用的字符处理装饰器类。

1. InputStreamReader 与 OutputStreamWriter

如果串流处理的字节数据，实际上代表某些字符的编码数据，而你想要将这些字节数据转换为对应的编码字符，可以使用 InputStreamReader、OutputStreamWriter 对串流数据打包。

在建立 InputStreamReader 与 OutputStreamWriter 时，可以指定编码，如果没有指定编码，则以 JVM 启动时所获取的默认编码来做字符转换。下面将 CharUtil 的 dump()改写，提供可指定编码的 dump()方法：

非卖品，仅供非商业用途或交流学习使用

Stream CharUtil2.java

```
package cc.openhome;

import java.io.*;

public class CharUtil2 {
    public static void dump(Reader src, Writer dest) throws IOException {
        try(src; dest) {
            char[] data = new char[1024];
            int length;
            while((length = src.read(data)) != -1) {
                dest.write(data, 0, length);
            }
        }
    }

    public static void dump(InputStream src, OutputStream dest,
                            String charset) throws IOException {
        dump(
            new InputStreamReader(src, charset),
            new OutputStreamWriter(dest, charset)
        );
    }

    // 采用默认编码
    public static void dump(InputStream src, OutputStream dest)
                            throws IOException {
        dump(src, dest, System.getProperty("file.encoding"));
    }
}
```

如果想以 UTF-8 处理字符数据，例如读取 UTF-8 的 Main.java 文本文件，并另存为 UTF-8 的 Main.txt 文本文件，则可以如下撰写：

```
CharUtil2.dump(
     new FileInputStream("Main.java"),
     new FileOutputStream("Main.txt"),
     "UTF-8"
);
```

2. BufferedReader 与 BufferedWriter

正如 BufferedInputStream、BufferedOutputStream 为 InputStream、OutputStream 提供缓冲区作用，以改进输入/输出的效率，BufferedReader、BufferedWriter 可对 Reader、Writer 提供缓冲区作用，在处理字符输入/输出时，对提高效率也会有所帮助。

举个使用 BufferedReader 的例子。在 JDK 1.4 之前，标准 API 并没有 Scanner 类，若要在文本模式下取得用户输入的字符串，要这样撰写：

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
String name = reader.readLine();
out.printf("Hello, %s!", name);
```

创建 BufferedReader 时要指定被打包的 Reader，可以指定或采用默认缓冲区大小。就 API 的使用而言，System.in 是 InputStream 实例，可以指定给 InputStreamReader 创建之用，

非卖品，仅供非商业用途或交流学习使用

InputStreamReader 是一种 Reader，所以可指定给 BufferedReader 创建之用。

就装饰器的作用而言，InputStreamReader 将 System.in 读入的字节数据做编码转换，而 BufferedReader 将编码转换后的数据做缓冲处理，以提高读取效率。BufferedReader 的 **readLine()** 方法，可以读取一行数据(以换行字符为依据)并以字符串返回，返回的字符串不包括换行字符。

3. PrintWriter

PrintWriter 与 PrintStream 使用上极为类似，不过除了可以对 OutputStream 打包之外，PrintWriter 还可以对 Writer 进行打包，提供 print()、println()、format()等方法。

10.3 重点复习

从应用程序角度来看，如果要将数据从来源读取，可以使用输入串流；如果要将数据写入目的地，可以使用输出串流。在 Java 中，输入串流代表对象为 java.io.InputStream 实例，输出串流代表对象为 java.io.OutputStream 实例。无论数据源或目的地为何，只要设法取得 InputStream 或 OutputStream 的实例，接下来操作输入/输出的方式都是一致的，无须理会来源或目的地的真正形式。

在不使用 InputStream 与 OutputStream 时，必须使用 close()方法关闭串流。由于 InputStream 与 OutputStream 操作了 java.io.Closeable 接口，其父接口为 java.lang.AutoCloseable 接口，因此可使用 JDK7 尝试自动关闭资源语法。

FileInputStream 是 InputStream 的子类，可以指定文件名创建实例，一旦创建文档就开启，就可用来读取数据。FileOutputStream 是 OutputStream 的子类，可以指定文件名创建实例，一旦创建文档就开启，就可以用来写出数据。无论 FileInputStream 还是 FileOutputStream，不使用时都用 close()关闭文档。

ByteArrayInputStream 是 InputStream 的子类，可以指定 byte 数组创建实例，一旦创建就可将 byte 数组当作数据源进行读取。ByteArrayOutputStream 是 OutputStream 的子类，可以指定 byte 数组创建实例，一旦创建即将 byte 数组当作目的地写出数据。

InputStream、OutputStream 提供串流基本操作，如果想要为输入/输出的数据做加工处理，则可以使用打包器类。常用的打包器有具备缓冲区作用的 BufferedInputStream、BufferedOutputStream，具备数据转换处理作用的 DataInputStream、DataOutputStream，具备对象串行化能力的 ObjectInputStream、ObjectOutputStream 等。

针对字符数据的读取，Java SE 提供了 java.io.Reader 类，其抽象化了字符数据读取的来源。针对字符数据的写入，则提供了 java.io.Writer 类，其抽象化了数据写出的目的地。

FileReader、FileWriter 则可以对文档做读取与写入，读取或写入时默认会使用操作系统默认编码来做字符转换。在启动 JVM 时，可以通过-Dfile.encoding 来指定 FileReader、FileWriter 所使用的编码。

Reader、Writer 也有一些装饰器类可供使用。如果串流处理的字节数据，实际上代表某些字符的编码数据，而你想要将这些字节数据转换为对应的编码字符，可以使用 InputStreamReader、OutputStreamWriter 对串流数据打包。BufferedReader、BufferedWriter 可

非卖品，仅供非商业用途或交流学习使用

对 Reader、Writer 提供缓冲区作用，在处理字符输入/输出时，对提高效率也会有所帮助。PrintWriter 与 PrintStream 使用上极为类似，不过除了可以对 OutputStream 打包之外，PrintWriter 还可以对 Writer 进行打包，提供 print()、println()、format()等方法。

10.4 课后练习

1. 在异常发生时，可以使用异常对象的 printStackTrace()显示堆栈追踪，如何改写以下程序，使得异常发生时，可将堆栈追踪附加至 UTF-8 编码的 exception.log 文档：

```
package cc.openhome;
import java.io.*;
public class Exercise1 {
    public static void dump(InputStream src, OutputStream dest)
                                    throws IOException {
        try (src; dest) {
            byte[] data = new byte[1024];
            int length = -1;
            while ((length = src.read(data)) != -1) {
                dest.write(data, 0, length);
            }
        } catch(IOException ex) {
            throw ex;
        }
    }

}
```

2. 请撰写程序：可将任何编码的文本文件读入，指定文档名转存为 UTF-8 的文本文件。

3. 试着撰写一个 FileUtil 类，其中包括 open()方法，包括了处理 FileInputStream 实例建立、close()方法调用，以及将 IOException 打包为 java.io.UncheckedIOException 实例并重新抛出的程序流程。FileUtil 的 open()方法应该要能如下使用：

```
package cc.openhome;

import java.util.Scanner;
import static cc.openhome.FileUtil.open;
public class Exercise3 {
    public static void main(String[] args) {
        open(args[0], fileInputStream -> {
            Scanner file = new Scanner(fileInputStream);
            while(file.hasNextLine()) {
                System.out.println(file.nextLine());
            }
        });
    }
}
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 11

线程与并行 API

学习目标

- 认识 Thread 与 Runnable
- 使用 synchronized
- 使用 wait()、notify()、notifyAll()
- 运用高级并行 API

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

11.1 线程

到目前为止介绍过的各种范例都是单线程程序，也就是启动的程序从 main() 程序进入点开始至结束只有一个流程。有时候需要设计程序可以拥有多个流程，也就是所谓的多线程(Multi-thread)程序。

11.1.1 线程简介

如果要设计一个龟兔赛跑游戏，赛程长度为 10 步，每经过一秒，乌龟会前进一步，兔子则可能前进两步或睡觉，那该怎么设计呢？如果用目前所学过的单线程程序来说，你可能会这样设计：

Thread TortoiseHareRace.java

```
package cc.openhome;

import static java.lang.System.out;

public class TortoiseHareRace {
    public static void main(String[] args) {
        boolean[] flags = {true, false};
        int totalStep = 10;
        int tortoiseStep = 0;
        int hareStep = 0;
        out.println("龟兔赛跑开始...");
        while(tortoiseStep < totalStep && hareStep < totalStep) {
            tortoiseStep++; ←— ❶ 乌龟走一步
            out.printf("乌龟跑了 %d 步...%n", tortoiseStep);
            boolean isHareSleep = flags[((int) (Math.random() * 10)) % 2]; ←❷ 随机睡觉
            if(isHareSleep) {
                out.println("兔子睡着了 zzzz");
            } else {
                hareStep += 2; ←— ❸ 兔子走两步
                out.printf("兔子跑了 %d 步...%n", hareStep);
            }
        }
    }
}
```

目前程序只有一个流程，就是从 main() 开始至结束的流程。tortoiseStep 递增 1 表示乌龟走一步❶，兔子则可能随机睡觉❷，如果不是睡觉就将 hareStep 递增 2，表示兔子走两步❸，只要乌龟或兔子其中一个走完 10 步就离开循环，表示比赛结束。

由于程序只有一个流程，所以只能将乌龟与兔子的行为混杂在这个流程中撰写。为什么每次都先递增乌龟再递增兔子步数呢？这样对兔子很不公平啊！如果可以撰写程序启动两个流程，一个是乌龟流程，一个兔子流程，程序逻辑会比较清楚。

在 Java 中，如果想在 main() 以外独立设计流程，可以撰写类操作 **java.lang.Runnable** 接口，流程的进入点是操作在 **run()** 方法中。例如，可以这样设计乌龟的流程：

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Thread Tortoise.java

```
package cc.openhome;

public class Tortoise implements Runnable {
    private int totalStep;
    private int step;

    public Tortoise(int totalStep) {
        this.totalStep = totalStep;
    }

    @Override
    public void run() {
        while (step < totalStep) {
            step++;
            System.out.printf("乌龟跑了 %d 步...%n", step);
        }
    }
}
```

在 Tortoise 类中，乌龟的流程会从 run() 开始，乌龟只要专心负责每秒走一步就可以了，不会混杂兔子的流程。同样地，可以这样设计兔子的流程：

Thread Hare.java

```
package cc.openhome;

public class Hare implements Runnable {
    private boolean[] flags = {true, false};
    private int totalStep;
    private int step;

    public Hare(int totalStep) {
        this.totalStep = totalStep;
    }

    @Override
    public void run() {
        while (step < totalStep) {
            boolean isHareSleep = flags[((int) (Math.random() * 10)) % 2];
            if (isHareSleep) {
                System.out.println("兔子睡着了 zzzz");
            } else {
                step += 2;
                System.out.printf("兔子跑了 %d 步...%n", step);
            }
        }
    }
}
```

在 Hare 类中，兔子的流程会从 run() 开始，兔子只要专心负责每秒睡觉或走两步就可以了，不会混杂乌龟的流程。

在 Java 中，从 main() 开始的流程会由主线程(Main Thread)执行，那么刚才设计的 Tortoise

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

与 Hare，run()方法定义的流程该由谁执行呢？可以创建 **Thread** 实例来执行 **Runnable** 实例定义的 **run()**方法。例如：

Thread TortoiseHareRace2.java

```
package cc.openhome;

public class TortoiseHareRace2 {
    public static void main(String[] args) {
        Tortoise tortoise = new Tortoise(10);
        Hare hare = new Hare(10);
        Thread tortoiseThread = new Thread(tortoise);
        Thread hareThread = new Thread(hare);
        tortoiseThread.start();
        hareThread.start();
    }
}
```

在这个程序中，主线程执行 main()定义的流程，main()定义的流程中建立了 tortoiseThread 与 hareThread 两个线程，这两个线程会分别执行 Tortoise 与 Hare 的 run()定义的流程，要启动线程执行指定流程，必须调用 Thread 实例的 **start()**方法。执行如下所示范例：

```
乌龟跑了 1 步...
兔子睡着了 zzzz
乌龟跑了 2 步...
兔子跑了 2 步...
乌龟跑了 3 步...
兔子跑了 4 步...
乌龟跑了 4 步...
兔子睡着了 zzzz
乌龟跑了 5 步...
兔子睡着了 zzzz
乌龟跑了 6 步...
兔子睡着了 zzzz
乌龟跑了 7 步...
兔子跑了 6 步...
乌龟跑了 8 步...
兔子跑了 8 步...
乌龟跑了 9 步...
兔子睡着了 zzzz
乌龟跑了 10 步...
兔子跑了 10 步...
```

11.1.2 Thread 与 Runnable

从抽象观点与开发者的角度来看，JVM 是台虚拟计算机，只安装一个称为主线程的 CPU，可执行 main()定义的执行流程。如果想要为 JVM 加装 CPU，就是创建 Thread 实例，要启动额外 CPU 就是调用 Thread 实例的 start()方法。额外 CPU 执行流程的进入点，可以定义在 Runnable 接口的 run()方法中。

提示 »» 实际上 JVM 启动后，不只有一个主线程，还会有垃圾收集、内存管理等线程，不过这是底层机制，就撰写程序的角度来看 JVM，确实只有一个主线程。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

除了将流程定义在 **Runnable** 的 **run()** 方法中之外，另一个撰写多线程程序的方式，就是继承 **Thread** 类，重新定义 **run()** 方法。单就撰写程序的角度来看，前面的龟兔赛跑也可以改写为以下所示：

```
public class TortoiseThread extends Thread {
    ...与 Tortoise 相同，故略

    @Override
    public void run() {
        ...与 Tortoise 相同，故略
    }
}

public class HareThread extends Thread {
    ...与 Hare 相同，故略

    @Override
    public void run() {
        ...与 Hare 相同，故略
    }
}
```

这两个类分别继承 Thread 重新定义 run() 方法，可以在 main() 主流程中，这样撰写程序启动线程执行额外流程：

```
new TortoiseThread(10).start();
new HareThread(10).start();
```

在 Java 中，任何线程可执行的流程都要定义在 Runnable 的 run() 方法。事实上，Thread 类本身也操作了 Runnable 接口，而 run() 方法的操作如下：

```
...
    @Override
    public void run() {
        if (target != null) {
            target.run();
        }
    }
...
```

调用 Thread 实例的 start() 方法后，会执行以上定义的 run() 方法，如果创建 Thread 时指定 Runnable 实例，就会由 target 参考。所以，如果直接继承 Thread 并重新定义 run() 方法，当然也可以执行其中流程。

那么是操作 Runnable 在 run() 中定义额外流程好，还是继承 Thread 在 run() 中定义额外流程好？操作 **Runnable** 接口的好处就是较有弹性，你的类还有机会继承其他类。若继承了 **Thread**，那该类就是一种 **Thread**，通常是为了直接利用 **Thread** 中定义的一些方法，才会继承 **Thread** 来操作。

在 JDK8 之后，由于可以使用 Lambda 表达式，而建构 Thread 时可以接受 Runnable 实例，在某些必须以匿名类语法建构 Thread 的场合，可以考虑用 Lambda 表达式来操作 Runnable，然后再用以建立 Thread。本来有个 Thread 的建立如下：

```
Thread someThread = new Thread() {
    public void run() {
        // 方法操作内容…
    }
};
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

可以改为以下较简洁的方式操作：

```
Thread someThread = new Thread(() -> {
    // 方法操作内容...
});
```

11.1.3 线程生命周期

线程生命周期颇为复杂，下面将从最简单的开始介绍。

1. Daemon 线程

主线程会从 `main()` 方法开始执行，直到 `main()` 方法结束后停止 JVM。如果主线程中启动了额外线程，默认会等待被启动的所有线程都执行完 `run()` 方法才中止 JVM。如果一个 `Thread` 被标示为 Daemon 线程，在所有的非 Daemon 线程都结束时，JVM 自动就会终止。

从 `main()` 方法开始的就是一个非 Daemon 线程，可以使用 **`setDaemon()`** 方法来设定一个线程是否为 Daemon 线程。以下是个简单示范，可以试着移除调用 `setDaemon()` 该行的批注，分析执行前后的差异。

Thread DaemonDemo.java

```
package cc.openhome;

public class DaemonDemo {
    public static void main(String[] args) {
        Thread thread = new Thread() {
            public void run() {
                while(true) {
                    System.out.println("Orz");
                }
            }
        };
        thread.setDaemon(true);
        thread.start();
    }
}
```

如果没有使用 `setDaemon()` 设定为 `true`，则程序会不断地输出 Orz 而不终止；使用 **`isDaemon()`** 方法可以判断线程是否为 Daemon 线程。

默认所有从 Daemon 线程产生的线程也是 Daemon 线程，因为基本上由一个背景服务线程衍生出来的线程，也应该是为了在背景服务而产生的，所以在产生它的线程停止时，也应该一并跟着停止。

2. Thread 基本状态图

在调用 `Thread` 实例 `start()` 方法后，基本状态为可执行(Runnable)、被阻断(Blocked)、执行中(Running)，状态间的转移如图 11.1 所示。

实例化 `Thread` 并执行 `start()` 之后，线程进入 Runnable 状态，此时线程尚未真正开始执行 `run()` 方法，必须等待排班器(Scheduler)排入 CPU 执行，线程才会执行 `run()` 方法，进入 Running 状态。线程看起来像是同时执行，但事实上在同一时间点上，一个 CPU 只能执行一个线程，

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

只是 CPU 会不断切换线程，且切换动作很快，所以看来像是同时执行。

线程有其优先权，可使用 Thread 的 **setPriority()** 方法设定优先权，可设定值为 1(**Thread.MIN_PRIORITY**) 到 10(**Thread.MAX_PRIORITY**)，默认是 5(**Thread.NORM_PRIORITY**)，超出 1 到 10 外的设定值会抛出 IllegalArgumentException。数字越大优先权越高，排班器越优先排入 CPU，如果优先权相同，则输流执行(Round-robin)。

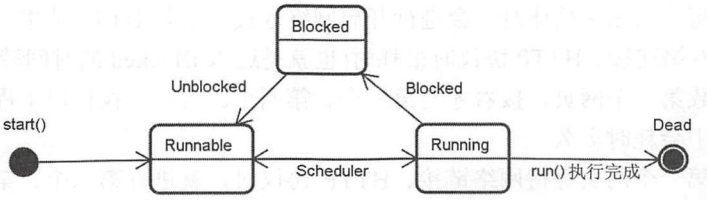


图 11.1 Thread 基本状态图

有几种状况会让线程进入 Blocked 状态，例如前面调用 Thread.sleep() 方法就会让线程进入 Blocked(其他还有进入 synchronized 前竞争对象锁定的阻断、调用 wait() 的阻断等，在之后的内容中介绍)；等待输入/输出完成也会进入 Blocked，之前章节范例文本模式下等待用户输入时就是实际的例子。运用多线程，当某线程进入 Blocked 时，让另一线程排入 CPU 执行(成为 Running 状态)，避免 CPU 空闲下来，经常是改进效能的方式之一。

例如以下这个程序可以指定网址下载网页，来看看不使用线程时花费的时间。

Thread Download.java

```
package cc.openhome;

import java.net.URL;
import java.io.*;

public class Download {
    public static void main(String[] args) throws Exception {
        URL[] urls = {
            new URL("http://openhome.cc/Gossip/Encoding/"),
            new URL("http://openhome.cc/Gossip/Scala/"),
            new URL("http://openhome.cc/Gossip/JavaScript/"),
            new URL("http://openhome.cc/Gossip/Python/")
        };

        String[] fileNames = {
            "Encoding.html",
            "Scala.html",
            "JavaScript.html",
            "Python.html"
        };

        for(int i = 0; i < urls.length; i++) {
            dump(urls[i].openStream(), new FileOutputStream(fileNames[i]));
        }
    }

    static void dump(InputStream src, OutputStream dest)throws IOException {
        try (src; dest) {
            byte[] data = new byte[1024];
            int length;
```


严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
            while ((length = src.read(data)) != -1) {
                dest.write(data, 0, length);
            }
        }
    }
}
```

这个程序在每一次 for 循环时，会进行开启网络连接、进行 HTTP 请求，然后再进行文档写入等，在等待网络连接、HTTP 协议时很耗时(也就是进入 Blocked 的时间较长)，第一个网页下载完后，再下载第二个网页，接着才是第三个、第四个。可以先执行以上程序，看看你的计算机在网络环境中会耗时多久。

如果可以在第一个网页等待网络链接、HTTP 协议时，就进行第二个、第三个、第四个网页的下载，那效率会提高很多。例如：

Thread Download2.java

```
package cc.openhome;

import java.net.URL;
import java.io.*;

public class Download2 {

    public static void main(String[] args) throws Exception {
        URL[] urls = {
            new URL("http://openhome.cc/Gossip/Encoding/"),
            new URL("http://openhome.cc/Gossip/Scala/"),
            new URL("http://openhome.cc/Gossip/JavaScript/"),
            new URL("http://openhome.cc/Gossip/Python/")
        };

        String[] fileNames = {
            "Encoding.html",
            "Scala.html",
            "JavaScript.html",
            "Python.html"
        };

        for (int i = 0; i < urls.length; i++) {
            int index = i;
            new Thread(() -> {
                try {
                    dump(urls[index].openStream(),
                        new FileOutputStream(fileNames[index]));
                } catch(IOException ex) {
                    throw new RuntimeException(ex);
                }
            }).start();
        }
    }

    static void dump(InputStream src, OutputStream dest)
                                throws IOException {
        ...同前一个范例，故略
    }
}
```


严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

这次的范例在 for 循环时，会建立新的 Thread 并启动，以进行网页下载。可以执行看看与上一个范例的差别有多少，这个范例花费的时间明显会少很多。

线程因输入/输出进入 Blocked 状态，在完成输入/输出后，会回到 Runnable 状态，等待排班器排入执行(Running)状态。一个进入 Blocked 状态的线程，可以由另一个线程调用该线程的 interrupt() 方法，让它离开 Blocked 状态。

举个例子来说，使用 **Thread.sleep()** 会让线程进入 Blocked 状态，若此时有其他线程调用该线程的 interrupt() 方法，会抛出 InterruptedException 异常对象，这是让线程“醒过来”的方式。以下是个简单示范：

Thread InterruptedDemo.java

```
package cc.openhome;

public class InterruptedDemo {
    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                try {
                    Thread.sleep(99999);
                } catch (InterruptedException ex) {
                    System.out.println("我醒了 XD");
throw new RuntimeException(ex);
                }
            }
        };
        thread.start();
        thread.interrupt(); // 主线程调用 thread 的 interrupt()
    }
}
```

执行结果：

```
我醒了 XD
Exception in thread "Thread-0" java.lang.RuntimeException:
java.lang.InterruptedException: sleep interrupted
    at cc.openhome/cc.openhome.InterruptedDemo.lambda$main$0(InterruptedDemo.java:11)
    at java.base/java.lang.Thread.run(Thread.java:844)
Caused by: java.lang.InterruptedException: sleep interrupted
```

提示 »» InterruptedException 设计为可处理的受检异常，捕捉后必须思考，如果线程是因为某些条件符合下被迫中断而离开 Blocked 状态，你应该做哪些收尾动作？例如清除线程目前使用的资源之类等。各种情境下的处理方式，可以参考 Dealing with Interrupted Exception 中的作法：http://www.ibm.com/developerworks/java/library/j-jtp05236/

3. 安插线程

如果 A 线程正在运行，流程中允许 B 线程加入，等到 B 线程执行完毕后再继续 A 线程流程，则可以使用 **join()** 方法完成这个需求。这就好比你手上有份工作正在进行，老板安插另一工作要求先做好，然后你再进行原本在进行的工作。

当线程使用 join() 加入至另一线程时，另一线程会等待被加入的线程工作完毕，然后再继续它的动作，join() 的意思表示将线程加入成为另一线程的流程中。

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Thread JoinDemo.java

```
package cc.openhome;

import static java.lang.System.out;

public class JoinDemo {

    public static void main(String[] args) throws InterruptedException {
        out.println("Main thread 开始...");

        Thread threadB = new Thread(() -> {
            out.println("Thread B 开始...");
            for (int i = 0; i < 5; i++) {
                out.println("Thread B 执行...");
            }
            out.println("Thread B 将结束...");
        });

        threadB.start();
        threadB.join();  // Thread B 加入 Main thread 流程

        out.println("Main thread 将结束...");
    }
}
```

程序启动后主线程就开始，在主线程中新建 threadB，并在启动 threadB 后，将之加入(join())主线程流程中，所以 threadB 会先执行完毕，主线程才会再继续原本的流程。执行结果如下：

```
Main thread 开始...
Thread B 开始...
Thread B 执行...
Thread B 执行...
Thread B 执行...
Thread B 执行...
Thread B 执行...
Thread B 将结束...
Main thread A 将结束...
```

如果程序中 threadB 没有使用 join()将之加入主线程流程中，则最后一行显示“Thread A 执行”的描述会先执行完毕(因为 threadB 使用了 sleep()，这让主线程有机会取得时间执行)。

有时候加入的线程可能处理太久，你不想无止境等待这个线程工作完毕，则可以在 join()时指定时间，如 join(10000)，这表示加入成为流程的线程至多可处理 10000 毫秒，也就是 10 秒，如果加入的线程还没执行完毕就不理它了，目前线程可继续执行原本工作流程。

4. 停止线程

线程完成 run()方法后，就会进入 Dead，进入 Dead(或已经调用过 start()方法)的线程不可以再次调用 start()方法，否则会抛出 IllegalThreadStateException。

Thread 类上定义有 stop()方法，不过被标示为 Deprecated，这表示过去确实定义过，后来因为会引发某些问题，为了确保向前兼容性，这些 API 没有直接剔除，但不建议新撰写的程序再使用它，如图 11.2 所示。

严禁网络传播本PDF，违者责任自负！

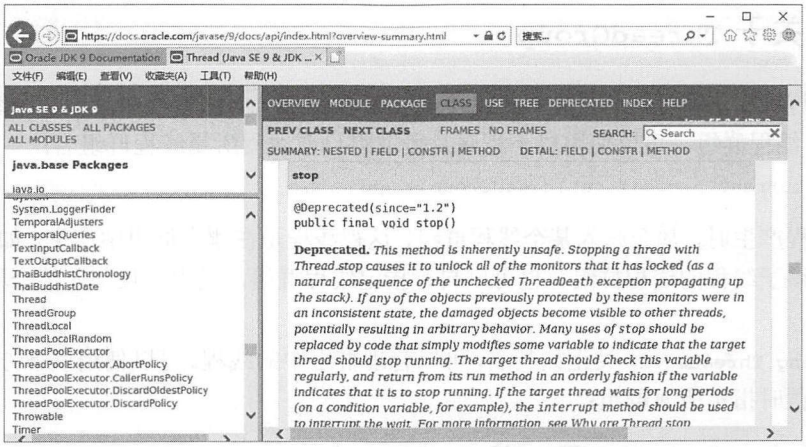


图 11.2　被标示为 Deprecated 的 API 不建议再使用

如果使用了被标示为 Deprecated 的 API，编译程序会提出警告，而在 IDE 中，通常会出现删除线表示不建议使用，在 NetBeans 中的样式如图 11.3 所示。

```
threadB.stop();
```

图 11.3　被标示为 Deprecated 的 API 会特别显示

直接调用 Thread 的 stop()方法，将不理会所设定的释放、取得锁定流程，线程会直接释放所有已锁定对象(锁定的概念稍后会介绍到)，这有可能使对象陷入无法预期状态。除了 stop()方法外，Thread 的 resume()、suspend()、destroy()等方法也不建议再使用，可参考以下的文件说明：

docs.oracle.com/javase/9/docs/api/java/lang/doc-files/threadPrimitiveDeprecation.html

如果要停止线程，最好自行操作，让线程跑完应有的流程，而非调用 Thread 的 stop()方法。例如，有个线程会在无穷循环中进行某个动作，那么停止线程的方式，就是让它有机会离开无穷循环：

```
public class Some implements Runnable {
    private boolean isContinue = true;
    ...
    public void stop() {
        isContinue = false;
    }

    public void run() {
        while(isContinue) {
            ...
        }
    }
}
```

在这个程序片段中，若线程执行了 run()方法，就会进入 while 循环，想要停止线程，就是调用 Some 的 stop()，这会将 isContinue 设为 false，在跑完此次 while 循环，下次 while 条件测试为 false 就会离开循环，执行完 run()方法，线程也就进入 Dead 状态而停止。

因此不仅有停止线程必须自行根据条件操作，线程的暂停、重启，也必须视需求操作，而不是直接调用 suspend()、resume()等方法。

严禁网络传播本PDF，违者责任自负！

11.1.4 关于 ThreadGroup

每个线程都属于某个线程群组(ThreadGroup)。若在 main()主流程中产生一个线程，该线程会属于 main 线程群组。可以使用以下程序片段取得目前线程所属线程群组名：

```
Thread.currentThread().getThreadGroup().getName();
```

每个线程产生时，都会归入某个线程群组，这视线程是在哪个群组中产生。如果没有指定，则归入产生该子线程的线程群组。也可以自行指定线程群组，线程一旦归入某个群组，就无法再更换。

java.lang.ThreadGroup 类正如其名，可以管理群组中的线程。可以使用以下方式产生群组，并在产生线程时指定所属群组：

```
ThreadGroup group1 = new ThreadGroup("group1");
ThreadGroup group2 = new ThreadGroup("group2");

Thread thread1 = new Thread(group1, "group1's member");
Thread thread2 = new Thread(group2, "group2's member");
```

ThreadGroup 的某些方法，可以对群组中所有线程产生作用。例如，**interrupt()**方法可以中断群组中所有线程，**setMaxPriority()**方法可以设定群组中所有线程最大优先权(本来就拥有更高优先权的线程不受影响)。

如果想要一次取得群组中所有线程，可以使用 **enumerate()**方法。例如：

```
Thread[] threads = new Thread[threadGroup1.activeCount()];
threadGroup1.enumerate(threads);
```

activeCount()方法取得群组的线程数量，enumerate()方法要传入 Thread 数组，这会将线程对象设定至每个数组索引。

ThreadGroup 中有个 **uncaughtException()**方法，群组中某个线程发生异常而未捕捉时，JVM 会调用此方法进行处理。如果 ThreadGroup 有父 ThreadGroup，就会调用父 ThreadGroup 的 **uncaughtException()**方法，否则看看异常是否为 ThreadDeath 实例。若是则什么都不做，若不是则调用异常的 printStrackTrace()。如果必须定义 ThreadGroup 中线程的异常处理行为，可以重新定义此方法。例如：

Thread ThreadGroupDemo.java

```
package cc.openhome;

public class ThreadGroupDemo {

    public static void main(String[] args) {
        ThreadGroup group = new ThreadGroup("group") {
            @Override
            public void uncaughtException(Thread thread, Throwable throwable) {
                System.out.printf("%s: %s%n",thread.getName(), throwable.getMessage());
            }
        };

        Thread thread = new Thread(group, () -> {
            throw new RuntimeException("测试异常");
        });
```

严禁网络传播本PDF，违者责任自负！

```
        thread.start();
    }
}
```

uncaughtException()方法第一个参数可取得发生异常的线程实例，第二个参数可取得异常对象，范例中显示了线程的名称及异常信息。结果如下所示：

```
Thread-0: 测试异常
```

在 JDK5 之后，如果 ThreadGroup 中的线程发生异常，**uncaughtException()**方法的处理顺序是：

- 如果 ThreadGroup 有父 ThreadGroup，就会调用父 ThreadGroup 的 **uncaughtException()** 方法。
- 否则，看看 Thread 是否使用 **setUncaughtExceptionHandler()**方法设定 **Thread.UncaughtExceptionHandler** 实例，有的话就会调用其 uncaughtException()方法。
- 如果没有使用，看看异常是否为 ThreadDeath 实例，若“是”则什么都不做，若“否”则调用异常的 printStrackTrace()。

未捕捉异常会由线程实例 setUncaughtExceptionHandler()设定的 Thread.UncaughtExceptionHandler 实例处理，之后是线程的 ThreadGroup.uncaughtException()，然后是默认的 Thread.UncaughtExceptionHandler。

所以对于线程本身未捕捉的异常，可以自行指定处理方式。例如：

Thread ThreadGroupDemo2.java

```
package cc.openhome;

public class ThreadGroupDemo2 {

    public static void main(String[] args) {
        ThreadGroup group = new ThreadGroup("group");

        Thread thread1 = new Thread(group, () -> {
            throw new RuntimeException("thread1 测试例外");
        });
        thread1.setUncaughtExceptionHandler((thread, throwable) -> {
            System.out.printf("%s: %s%n",thread.getName(), throwable.getMessage());
        });

        Thread thread2 = new Thread(group, () -> {
            throw new RuntimeException("thread2 测试异常");
        });

        thread1.start();
        thread2.start();
    }
}
```

在这个范例中，t1、t2 都属于同一个 ThreadGroup，t1 设定了 Thread.UncaughtExceptionHandler 实例，所以未捕捉的异常会以 Thread.UncaughtExceptionHandler 定义的方式处理，t2 没有设定 Thread.UncaughtExceptionHandler 实例，所以由 ThreadGroup 默认的第三个处理方式，显示堆栈追踪。执行结果如下：

严禁网络传播本PDF，违者责任自负！

```
Exception in thread "Thread-1" java.lang.RuntimeException: thread2 测试异常
at cc.openhome.ThreadGroupDemo2.lambda$main$2(ThreadGroupDemo2.java:17)
at cc.openhome.ThreadGroupDemo2$$Lambda$3/2055281021.run(Unknown Source)
at java.lang.Thread.run(Thread.java:744)
Thread-0: thread1 测试异常
```

11.1.5 synchronized 与 volatile

还记得 6.2.5 节曾经开发过一个 ArrayList 类吗？在还没有接触多线程前，也就是将 ArrayList 用在只有主线程的环境中时，它没有什么问题。如果有两个以上的线程同时使用它会如何？例如：

Thread ArrayListDemo.java

```
package cc.openhome;

public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        Thread thread1 = new Thread(() -> {
            while (true) {
                list.add(1);
            }
        });

        Thread thread2 = new Thread(() -> {
            while (true) {
                list.add(2);
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

在这个范例中建立了两个线程，分别在 while 循环中对同一个 ArrayList 实例进行 add()。如果尝试执行程序，“有可能”会发生 ArrayIndexOutOfBoundsException 异常：

```
Exception in thread "Thread-1" java.lang.ArrayIndexOutOfBoundsException: 64
at cc.openhome.ArrayList.add(ArrayList.java:21)
at cc.openhome.ArrayListDemo.lambda$main$1(ArrayListDemo.java:15)
at cc.openhome.ArrayListDemo$$Lambda$2/1072591677.run(Unknown Source)
at java.lang.Thread.run(Thread.java:744)
```

这是概率问题，有可能发生，也有可能没发生，就因数组长度过长，JVM 分配到的内存不够，而发生 java.lang.OutOfMemoryError。此处将焦点放在为何会发生 ArrayIndexOutOfBoundsException 异常。来看 ArrayList 的 add()方法片段：

```
...
    public void add(Object o) {
        if(next == elems.length) {
            elems = Arrays.copyOf(elems, elems.length * 2);
        }
```

```
            elems[next++] = o;
        }
    ...
```

如果某线程调用了 add()方法，而此时 next 刚好等于内部数组长度，应该建立新数组并完成元素复制动作后，再在新数组中参考新 add()的元素，接着递增 next。也就是说，add()方法的流程应该一气呵成。

若有 t1、t2 两个线程同时调用 add()方法，假设 t1 执行 add()已经到了 list[next++] = o 该行，此时 CPU 排班器将 t1 置回 Runnable 状态，将 t2 置入 Running 状态，而 t2 执行 add()已完成 list[next++] = o 行的执行，此时 next 刚好等于数组长度。若此时 CPU 排班器将 t2 置回 Runnable 状态，将 t1 置入 Running 状态，于是 t1 开始 list[next++] = o 行，因为 next 刚好等于数组长度，结果就发生 ArrayIndexOutOfBoundsException。

这就是线程存取同一对象相同资源时所引发的竞速情况(Race Condition)，以此例而言，是因为 t1、t2 会同时存取 next，使得 next 在巧合情况下，脱离原本应管控的条件，我们称 ArrayList 这样的类为不具备线程安全(Thread-safe)的类。

1. 使用 synchronized

该怎么解决呢？可以在 add()方法上加上 **synchronized** 关键字。例如：

```
...
    public synchronized void add(Object o) {
        if(next == list.length) {
            list = Arrays.copyOf(list, list.length * 2);
        }
        list[next++] = o;
    }
...
```

在加上 synchronized 关键字之后，再次执行前面范例，就不会看到 ArrayIndexOutOfBounds-Exception 了，原因为何？

每个对象都会有个内部锁定(Intrinsic lock)，或称为监控锁定(Monitor lock)。被标示为 synchronized 的区块将会被监控，任何线程要执行 synchronized 区块都必须先取得指定的对象锁定。如果 A 线程已取得对象锁定开始执行 synchronized 区块，B 线程也想执行 synchronized 区块，会因无法取得对象锁定而进入等待锁定状态，直到 A 线程释放锁定(例如执行完 synchronized 区块)，B 线程才有机会取得锁定而执行 synchronized 区块。

如果在方法上标示 **synchronized**，则执行方法必须取得该实例的锁定。所以一旦 add()方法加上了 synchronized，t1 调用 add()时，就会取得对象锁定，若此时 t2 也想调用 add()方法，会因无法取得锁定而进入等待锁定状态，直到 t1 执行完 add()离开了 synchronized 区块释放锁定，t2 在取得锁定后，才可以调用 add()方法。简单来说，可以确保某线程执行 add()时，将 add()中定义的流程完整执行，从而避免了 ArrayIndexOutOfBoundsException 发生。

前面在说明图 11.1 中的 Blocked 状态时，是以 Thread.sleep()与输入/输出为例，实际上等待对象锁定时，也会进入 Blocked 状态，如图 11.4 所示。

非卖品，仅供非商业用途或交流学习使用

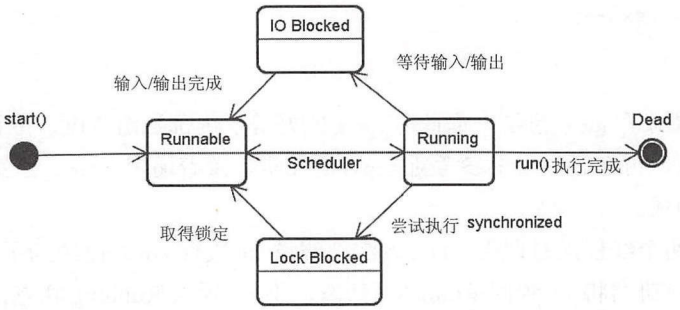


图 11.4　加上锁定条件的线程状态图

线程若因尝试执行 synchronized 区块而进入 Blocked，在取得锁定之后，会先回到 Runnable 状态，等待 CPU 排班器排入 Running 状态。

synchronized 不仅可以声明在方法上，也可以描述句方式使用。例如：

```
...
    public void add(Object o) {
        synchronized(this) {
            if(next == list.length) {
                list = Arrays.copyOf(list, list.length * 2);
            }
            list[next++] = o;
        }
    }
...
```

这个程序片段的意思就是，在线程要执行 **synchronized** 区块时，必须取得括号中指定的对象锁定。这个片段仅示范出语法，事实上此语法的目的之一，可应用于不想锁定整个方法，而只想锁定会发生竞速状况的区块，在执行完区块后线程即释放锁定，其他线程就有机会再竞争对象锁定，相较于将整个方法声明为 synchronized 来说，效率较高。

由于这个语法可指定取得锁定的对象来源，对于本身设计时没有考虑竞速问题的 API 来说，也可以这样撰写：

```
...
ArrayList list = new ArrayList();
Thread thread1 = new Thread(() -> {
    while(true) {
        synchronized(list) {
            list.add(1);
        }
    }
});

Thread thread2 = new Thread(() -> {
    while(true) {
        synchronized(list) {
            list.add(2);
        }
    }
});
...
```

非卖品，仅供非商业用途或交流学习使用

如果 ArrayList 的 add()方法本身没有标示 synchronized，像上面那样撰写的话，每次要执行粗体字区块时，必须取得 list 参考的对象锁定，因而也能确保 add()执行完成，避免 t1、t2 同时调用 add()方法而引发竞速问题。

第 9 章介绍过的 Collection 与 Map，都未考虑线程安全，可以使用 Collections 的 synchronizedCollection()、synchronizedList()、synchronizedSet()、synchronizedMap()等方法，这些方法会将传入的 Collection、List、Set、Map 操作对象打包，返回具线程安全的对象。例如，以下 List 操作：

```
List<String> list = new ArrayList<>();
synchronized(list) {
    ...
    list.add("...");
}
...
synchronized(list) {
    ...
    list.remove("...");
}
```

则可以这样加以简化：

```
List<String> list = Collections.synchronizedList(new ArrayList<String>());
...
list.add("...");
...
list.remove("...");
```

使用 synchronized 描述句，可以做到更细致控制。例如，提供不同对象作为锁定来源：

```
public class Material {
    private int data1 = 0;
    private int data2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void doSome() {
        ...
        synchronized(lock1) {
            ...
            data1++;
            ...
        }
        ...
    }

    public void doOther() {
        ...
        synchronized(lock2) {
            ...
            data2--;
            ...
        }
        ...
    }
}
```

在这里想避免 doSome()中同时被两个以上线程执行 synchronized 区块，或是 doOther()中

非卖品，仅供非商业用途或交流学习使用

被两个以上的线程同时执行 syn chronized 区块，但 data1 与 data2 并不同时出现在两个方法中，所以有个线程执行 doSome()而另一个线程执行 doOther()时，并不会引发共享存取问题。此时分别提供不同对象作为锁定来源，就不会导致 doSome()中 synchronized 被线程存取时，doOther()中 synchronized 被另一个试图存取时所引发的阻断延迟。

Java 的 synchronized 提供的是可重入同步(Reentrant Synchronization)，也就是线程取得某对象锁定后，若执行过程中又要执行 synchronized，尝试取得锁定的对象来源又是同一个，则可以直接执行。

由于线程无法取得锁定时会造成阻断，不正确地使用 synchronized 有可能造成效能低下，另一问题则是死结(Dead Lock)。例如，有些资源在多线程下彼此交叉取用，就有可能造成死结。以下是个简单的例子：

Thread DeadLockDemo.java

```
package cc.openhome;

class Resource {
    private String name;
    private int resource;

    Resource(String name, int resource) {
        this.name = name;
        this.resource = resource;
    }

    String getName() {
        return name;
    }

    synchronized int doSome() {
        return ++resource;
    }

    synchronized void cooperate(Resource resource) {
        resource.doSome();
        System.out.printf("%s 整合 %s 的资源%n",
                          this.name, resource.getName());
    }
}

public class DeadLockDemo {
    public static void main(String[] args) {
        Resource resource1 = new Resource("resource1", 10);
        Resource resource2 = new Resource("resource2", 20);

        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                resource1.cooperate(resource2);
            }
        });
        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                resource2.cooperate(resource1);
            }
        });
```


非卖品，仅供非商业用途或交流学习使用

```
        thread1.start();
        thread2.start();
    }
}
```

上面这个程序会不会发生死结，也是概率问题。可以尝试执行看看，有时程序可顺利执行完成，有时程序会整个停顿。

会发生死结的原因在于，t1 在调用 resource1.corportate(resource2)时，会取得 resource1 的锁定，若此时 t2 正好也调用 resource2.corportate(resource1)，会取得 resource2 的锁定，凑巧 t1 现在打算运用传入的 resource2 调用 doSome()，理应取得 resource2 的锁定，但锁定现在被 t2 拿走了，于是 t1 进入阻断，而 t2 也打算运用传入的 resource1 调用 doSome()，理应取得 resource1 的锁定，但锁定现在是 t1 取走，于是 t2 也进入等待。

要更简单解释这个范例为何有时会死结，就是偶尔会发生两个线程都处于“你不放开 resource1 锁定，我就不放开 resource2 锁定”的状态，Java 在死结发生时陷入停顿状态，所以必须在程序设计时避免死结的发生。

2. 使用 volatile

synchronized 要求达到所标示区块的互斥性(Mutual Exclusion)与可见性(Visibility)，互斥性是指 synchronized 区块同时间只能有一个线程，可见性是指线程离开 synchronized 区块后，另一线程接触到的就是上一线程改变后的对象状态。

提示»» Volatile 的概念较深，初学者可以略过。

在 Java 中对于可见性的要求，可以使用 volatile 达到变量范围。在讨论变量可见性前，先来看以下范例：

Thread Variable1Test.java

```
package cc.openhome;

class Variable1 {
    static int i = 0, j = 0;

    static void one() {
        i++;
        j++;
    }

    static void two() {
        System.out.printf("i = %d, j = %d%n", i, j);
    }
}

public class Variable1Test {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            while (true) {
                Variable1.one();
            }
        });
        Thread thread2 = new Thread(() -> {
```


非卖品，仅供非商业用途或交流学习使用

```
            while (true) {
                Variable1.two();
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

thread1 会调用 Variable1.one()，thread2 会调用 Variable1.two()，有可能会发生 thread2 调用 Variable1.two() 取得 i 值后，切换至 thread1 不断执行 Variable1.one() 多次，再切回 thread2 取得 j 值，因此有可能出现 j "远大于" i 的显示结果：

```
...
i = 256531535, j = 256531550
i = 256539063, j = 256539076
i = 256546975, j = 256546989
i = 256554509, j = 256554524
i = 256561967, j = 256561981
i = 256561967, j = 256561981
...
```

可以在 one() 与 two() 方法上标示 synchronized，这样每次 thread1 调用 one() 时，thread2 就必须等待 thread1 释放锁定，才能调用 two()，thread2 调用 two() 时，thread1 就必须等待 thread2 释放锁定，才可以调用 one()。如：

Thread Variable2Test.java

```
package cc.openhome;

class Variable2 {
    static int i = 0, j = 0;

    static synchronized void one() {
        i++;
        j++;
    }

    static synchronized void two() {
        System.out.printf("i = %d, j = %d%n", i, j);
    }
}

public class Variable2Test {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            while (true) {
                Variable2.one();
            }
        });
        Thread thread2 = new Thread(() -> {
            while (true) {
                    Variable2.two();
                }
        });
```


非卖品，仅供非商业用途或交流学习使用

```
        thread1.start();
        thread2.start();
    }
}
```

这保证了 one() 中 i 与 j 一定都会递增完成，才释放锁定，或是 two() 中一定取得 i 与 j 值显示后，才释放锁定。所以执行结果中，显示 i 与 j 值一定都是“相同”：

```
...
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
i = 95147, j = 95147
...
```

由于 synchronized 会造成线程阻断，使得调用 one() 时，其他线程就不能调用 two()，反之亦然，所以执行时会看到速度明显变慢。

在不加上 synchronized 的时候，为何会出现 j 值大于 i 值的情况？为了效率，线程可以快取变量的值，在 thread2 调用 Variable1.two() 从共享内存中的 i 变量取得值后，会快取于自己的内存空间中，如果此时切换至 thread1 不断执行 Variable1.one() 多次，共享内存中的 i 值已被变更多次，再切回 thread2 取得 j 值在自己的内存空间，然后将自己内存空间中快取的值输出，因此有可能出现 j 大于 i 的显示结果。

在变量上声明 **volatile**，表示变量是不稳定、易变的，也就是可能在多线程下存取，这保证变量的可见性，也就是若有线程变动了变量值，另一线程一定可看到变更。被标示为 volatile 的变量，不允许线程快取，变量值的存取一定是在共享内存中进行。举例来说，若将前面范例的 i 与 j 声明为 volatile：

Thread Variable3Test.java

```
package cc.openhome;

class Variable3 {
    volatile static int i = 0, j = 0;

    static void one() {
        i++;
        j++;
    }

    static void two() {
        System.out.printf("i = %d, j = %d%n", i, j);
    }
}

public class Variable3Test {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            while (true) {
                Variable3.one();
            }
        });
```


非卖品，仅供非商业用途或交流学习使用

```
        Thread thread2 = new Thread(() -> {
            while (true) {
                Variable3.two();
            }
        });
        thread1.start();
        thread2.start();
    }
}
```

这个范例在执行时，大部分情况是 i 值等于 j 值，或是 i 值大于 j 值，对于 j 远大于 i 值的情况则减少许多：

```
...
i = 13235787, j = 13235787
i = 13236353, j = 13236353
i = 13236905, j = 13236906
i = 13237459, j = 13237459
i = 13238014, j = 13238013
i = 13471968, j = 13471968
i = 13471968, j = 13471968
i = 13471968, j = 13471968
i = 13471968, j = 13471968
i = 13471968, j = 13471968
...
```

例如，若 thread2 调用 two() 至取得 i 值，由于 i 是 volatile，所以不会快取，接着切换至 thread1 调用 one() 多次，然后切回 thread2 继续前面 two() 的流程，由于 volatile 要保证 thread1 对 i 的变更对 thread2 一定可见，所以 thread2 必须从共享内存中取得变更后的 i 值，然后再取得 j 值，此时看到的情况就是 i 等于 j 值。

若 thread1 调用 one() 至递增 i 值后，切换至 thread2 调用 two() 取得 i 与 j 显示，则显示的就是 i 大于 j 的结果。至于 j 会大于 i 的情况，则是发生在已将 i 值指定给 System.out.printf() 的第二个参数，而此时切换至 t1 调用 one() 完成，后来又切换至 thread2 继续 two() 流程，此时又将 j 指定给 System.out.printf() 的第三个参数，因此造成 j 大于 i 的结果。但这种情况发生的机会，比起没有标示 volatile 少了许多。

由这三个范例可见，**volatile** 保证的是单一变数的可见性，线程对变量的存取一定是在共享内存中，不会在自己的内存空间中快取变量，线程对共享内存中变量的存取，另一线程一定看得到。

就 Variable3Test 来看，如果 two() 显示时，想保证 i 值一定等于 j 值，使用 volatile 并不是正确的用法，而应该如 Variable2Test 中使用 synchronized。这有几个原因：首先递增(递减)运算符实际上是取得变量值、运算、设定变量值等动作，volatile 只保证 thread1 对 i 或 j 变量值的设定，thread2 一定可见；one() 中包括了 i++、j++两个描述，要保证这两个描述执行完，本来就该用 synchronized；two() 中包括了取得 i、j 变量值及执行 System.out.printf() 等动作，要保证这些描述执行完毕，也是 synchronized 的职责。

以下是个正确使用 volatile 的例子：

```
public class Some implements Runnable {
    private volatile boolean isContinue = true;
```


非卖品，仅供非商业用途或交流学习使用

```
    ...
    public void stop() {
        isContinue = false;
    }

    public void run() {
        while(isContinue) {
            ...
        }
    }
}
```

如果有 thread1 线程正在执行 Some 实例 run()中的 while 循环，你不希望 thread1 因快取了 isContinue，使得 thread2 调用 stop()方法设定 isContinue 为 false，而 thread1 无法实时在下次 while 条件检查时看到 thread2 对 isContinue 的变更，就可以将 isContinue 标示为 volatile。

提示»» 以下链接是关于 volatile 更高级的介绍，也有几个正确使用 volatile 以及不正确使用 volatile 的例子：
http://www.ibm.com/developerworks/java/library/j-jtp06197/

11.1.6 等待与通知

wait()、**notify()**与 **notifyAll()**是 Object 定义的方法，可以通过这 3 个方法控制线程释放对象的锁定，或者通知线程参与锁定竞争。

前面介绍过，线程进入 synchronized 范围前，要先取得指定对象的锁定。执行 synchronized 范围的程序代码期间，若调用锁定对象的 wait()方法，线程会释放对象锁定，并进入对象等待集合(Wait Set)而处于阻断状态，其他线程可以竞争对象锁定，取得锁定的线程可以执行 synchronized 范围的程序代码。

放在等待集合的线程不会参与 CPU 排班，wait()可以指定等待时间，时间到之后线程会再次加入排班，如果指定时间为 0 或不指定，则线程会持续等待，直到被中断(调用 interrupt())或是告知(notify())可以参与排班。

被竞争锁定的对象在调用 notify()时，会从对象等待集合中随机通知一个线程加入排班，再次执行 synchronized 前，被通知的线程会与其他线程共同竞争对象锁定；如果调用 notifyAll()，所有等待集合中的线程都会被通知参与排班，这些线程会与其他线程共同竞争对象锁定，如图 11.5 所示。

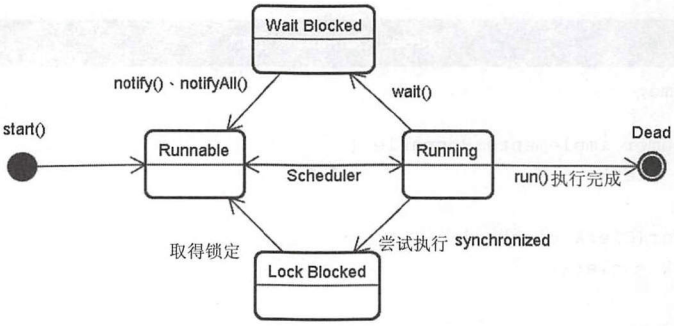


图 11.5 wait()、notify()、notifyAll()与 synchronized 线程状态图

简单地说，线程调用对象 wait()方法时，会先让出 synchronized 区块使用权并等待通知，

非卖品，仅供非商业用途或交流学习使用

或是等待指定时间，直到被 notify()或时间到时，(取得对象锁定之后)再从调用 wait()处开始执行，这就好比你叫柜台做事，做到一半时柜台人员叫你到等待区等候通知(或等候 1 分钟之类的)，当你被通知(或时间到时)，柜台人员才会继续为你服务。如果有多人同时等待，调用 notifyAll()就相当于通知等待区所有人，看谁先抢到柜台第一名，就先处理谁的工作。

说明 wait()、notify()或 notifyAll()应用常见范例，就是生产者与消费者。生产者会将生产的产品交给店员，而消费者从店员处取走产品消费，但店员一次只能储存固定数量产品。若生产者生产速度较快，店员可储存产品的量已满，店员会叫生产者等一下(wait)，如果有空位放产品了再通知(notify)生产者继续生产；如果消费者速度较快，将店中产品消费完毕了，店员会告诉消费者等一下(wait)，如果店中有产品了再通知(notify)消费者前来消费。

举个最简单的范例，假设生产者每次生产一个 int 整数交给店员：

Thread Producer.java

```
package cc.openhome;

public class Producer implements Runnable {
    private Clerk clerk;

    public Producer(Clerk clerk) {
        this.clerk = clerk;
    }

    public void run() {
        System.out.println("生产者开始生产整数...");
        for(int product = 1; product <= 10; product++) {  <-- ❶ 产生 1 到 10 的整数
            try {
                clerk.setProduct(product);  <-- ❷ 将产品交给店员
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

程序中使用 for 循环产生 1 到 10 的整数❶，Clerk 代表店员，可通过 setProduct()方法，将生产的整数交给店员❷。

消费者从店员处取走 int 整数：

Thread Consumer.java

```
package cc.openhome;

public class Consumer implements Runnable {
    private Clerk clerk;

    public Consumer(Clerk clerk) {
        this.clerk = clerk;
    }

    public void run() {
        System.out.println("消费者开始消耗整数...");
        for(int i = 1; i <= 10; i++) {  <-- ❶ 消费 10 次整数
```


非卖品，仅供非商业用途或交流学习使用

```
            try {
                clerk.getProduct(); ◀── ❷ 从店员处取走产品
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

程序中使用 for 循环来消费 10 次整数❶，可通过 Clerk 的 getProduct() 方法，从店员处取走整数❷。

由于店员一次只能持有一个 int 整数，所以必须尽到要求等待与通知的职责：

Thread Clerk.java

```
package cc.openhome;

public class Clerk {
    private int product = -1; ◀── ❶ 只持有一个产品，-1 表示没有产品

    public synchronized void setProduct(int product)
                                    throws InterruptedException {
        waitIfFull();    ◀── ❷ 看看店员有没有空间收产品，没有的话就等候
        this.product = product; ◀── ❸ 店员收货
        System.out.printf("生产者设定 (%d)%n", this.product);
        notify();  ◀── ❹ 通知等待集合中的线程(例如消费者)
    }

    private synchronized void waitIfFull() throws InterruptedException {
        while (this.product != -1) {
            wait();
        }
    }

    public synchronized int getProduct() throws InterruptedException {
        waitIfEmpty(); ◀── ❺ 看看目前店员有没有货，没有的话就等候
        int p = this.product; ◀── ❻ 准备交货
        this.product = -1; ◀── ❼ 表示货品被取走
        System.out.printf("消费者取走 (%d)%n", p);
        notify(); ◀── ❽ 通知等待集合中的线程(例如生产者)
        return p; ◀── ❾ 交货了
    }

    private synchronized void waitIfEmpty() throws InterruptedException {
        while (this.product == -1) {
            wait();
        }
    }
}
```

Clerk 只能持有一个整数，-1 表示目前没有产品❶。假设现在 Producer 调用了 setProduct()，此时不会进入 while 循环，所以设定 Clerk 的 product 为指定的整数❸，由于此时等待集合中没有线程，所以调用 notify() 没有作用❹。假设 Producer 再次调用 setProduct()，由于 Clerk 的 product 不为-1，表示店员无法收货了，于是进入 while 循环，执行了 wait()❷，于是 Producer

释放锁定，进入对象等待集合。

假设 Consumer 调用了 getProduct()，由于 Clerk 的 product 不为-1，所以不会进入 while 循环，于是 Clerk 准备交货❻，并将 product 设为-1❼，表示货品被取走，接着调用 notify()通知等待集合中的线程可以参与锁定竞争❽，最后将 p 返回并释放锁定❾。如果 Consumer 又调用了 getProduct()，由于 Clerk 的 product 为-1，表示没有产品了，于是进入 while 循环，执行 wait()后释放锁定进入对象等待集合❺。若此时 Producer 取得锁定，于是从 getProduct()中 wait()处继续执行。

注意»» 多个 Producer、Consumer 同时调用 Clerk 的 getProduct()、setProduct()，而 wait()没有放在条件式成立的循环中执行会如何呢？Java 规格书中说明，线程也有可能在未经 notify()、interrupt()或逾时情况下私自苏醒(Spurious Wakeup)，应用程序应考虑这种情况，wait()一定要在条件式成立的循环中执行。

可以使用以下程序来示范 Producer、Consumer 与 Clerk：

Thread ProducerConsumerDemo.java

```
package cc.openhome;

public class ProducerConsumerDemo {
    public static void main(String[] args) {
        Clerk clerk = new Clerk();
        new Thread(new Producer(clerk)).start();
        new Thread(new Consumer(clerk)).start();
    }
}
```

生产者会生产 10 个整数，而消费者会消耗 10 个整数，虽然生产与消费的速度不一，由于店员处只能放置一个整数，所以只能每生产一个才消耗一个：

```
生产者开始生产整数......
消费者开始消耗整数......
生产者设定 (1)
消费者取走 (1)
生产者设定 (2)
消费者取走 (2)
生产者设定 (3)
消费者取走 (3)
生产者设定 (4)
消费者取走 (4)
生产者设定 (5)
消费者取走 (5)
生产者设定 (6)
消费者取走 (6)
生产者设定 (7)
消费者取走 (7)
生产者设定 (8)
消费者取走 (8)
生产者设定 (9)
消费者取走 (9)
生产者设定 (10)
消费者取走 (10)
```

11.2 并行 API

使用 Thread 建立多线程程序，必须亲自处理 synchronized、对象锁定、wait()、notify()、notifyAll()等细节。如果需要的是线程池、读写锁等高级操作，从 JDK5 之后提供了 **java.util.concurrent** 包，基于其中的 API 可建立更稳固的并行应用程序。

11.2.1 Lock、ReadWriteLock 与 Condition

synchronized 要求线程必须取得对象锁定，才可执行所标示的区块范围，然而使用 synchronized 有许多限制，未取得锁定的线程会直接被阻断。如果你希望的功能是线程可尝试取得锁定，无法取得锁定时就先做其他事，直接使用 synchronized 必须通过一些设计才可完成这个需求，若要搭配 wait()、notify()、notifyAll()等方法，在设计上将会更为复杂。

java.util.concurrent.locks 包中提供 **Lock**、**ReadWriteLock**、**Condition** 接口以及相关操作类，可以提供类似 synchronized、wait()、notify()、notifyAll()的作用，以及更多高级功能。

提示>>> java.util.concurrent.locks 包中的 API 架构很简单，读者可以自行参考在线 API 文件。

1. 使用 Lock

Lock 接口主要操作类之一为 **ReentrantLock**，可以达到 synchronized 的作用，也提供额外的功能。先来看看如何使用 ReentrantLock 改写 6.2.5 节中的 ArrayList 为具线程安全的类。

Concurrency ArrayList.java

```
package cc.openhome;

import java.util.Arrays;
import java.util.concurrent.locks.*;

public class ArrayList<E> {
    private Lock lock = new ReentrantLock(); ←— ❶ 使用 ReentrantLock
    private Object[] elems;
    private int next;

    public ArrayList(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList() {
        this(16);
    }

    public void add(E elem) {
        lock.lock(); ←— ❷ 进行锁定
        try {
            if (next == elems.length) {
                elems = Arrays.copyOf(elems, elems.length * 2);
            }
            elems[next++] = elem;
        } finally {
            lock.unlock(); ←— ❸ 解除锁定
        }
    }
```

```
    public E get(int index) {
        lock.lock();
        try {
            return (E) elems[index];
        } finally {
            lock.unlock();
        }
    }

    public int size() {
        lock.lock();
        try {
            return next;
        } finally {
            lock.unlock();
        }
    }
}
```

ReentrantLock 顾名思义，如果已经有线程取得 Lock 对象锁定，尝试再次锁定同一 Lock 对象是可以的❶。想要锁定 Lock 对象，可以调用其 lock()方法❷，只有取得 Lock 对象锁定的线程，才可以继续往后执行程序代码，要解除锁定，可以调用 Lock 对象的 unlock()❸。

注意»» 为了避免调用 Lock 对象的 lock()后，在后续执行流程中抛出异常而无法解除锁定，一定要在 finally 中调用 Lock 对象的 unlock()方法。

Lock 接口还定义了 **tryLock()**方法，如果线程调用 tryLock()可以取得锁定会返回 true，若无法取得锁定并不会发生阻断，而是返回 false。来试着使用 tryLock()解决 11.1.5 节中 DeadLockDemo 的死结问题：

Concurrency NoDeadLockDemo.java

```
package cc.openhome;

import java.util.concurrent.locks.*;

class Resource {
    private ReentrantLock lock = new ReentrantLock();
    private String name;

    Resource(String name) {
        this.name = name;
    }

    void cooperate(Resource res) {
        while (true) {
            try {
                if (lockMeAnd(res)) {
                    System.out.printf("%s 整合 %s 的资源%n", this.name, res.name);
                    break;
                }
            } finally {
                unLockMeAnd(res);
            }
        }
    }
```

❶ 取得目前与传入的 Resource 的 Lock 锁定

❷ 如果两个 Resource 的 Lock 都取得锁定，才执行资源整合

❸ 资源整合成功，离开循环

❹ 解除目前与传入的 Resource 的 Lock 锁定

```
    private boolean lockMeAnd(Resource res) {
        return this.lock.tryLock() && res.lock.tryLock();
    }

    private void unLockMeAnd(Resource res) {
        if (this.lock.isHeldByCurrentThread()) {
            this.lock.unlock();
        }
        if (res.lock.isHeldByCurrentThread()) {
            res.lock.unlock();
        }
    }
}

public class NoDeadLockDemo {

    public static void main(String[] args) {
        Resource res1 = new Resource("resource1");
        Resource res2 = new Resource("resource2");

        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                res1.cooperate(res2);
            }
        });
        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                res2.cooperate(res1);
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

前面 DeadLockDemo 之所以会发生死结，在于两个线程在执行 cooperate()方法取得目前 Resource 锁定后，尝试调用另一 Resource 的 doSome()，因无法取得另一 Resource 的锁定而阻断。也就是说，线程因无法同时取得两个 Resource 的锁定而阻断。既然如此，在无法同时取得两个 Resource 的锁定时，干脆释放已取得的锁定，就可以解决问题。

改写后的 cooperate()会在 while 循环中，执行 lockMeAnd(res)❶，在该方法中使用目前 Resource 的 Lock 的 tryLock()尝试取得锁定，以及被传入 Resource 的 Lock 的 tryLock()尝试取得锁定，只有在两次 tryLock()返回值都是 true，也就是两个 Resource 都取得锁定后，才进行资源整合❷并离开 while 循环❸，无论哪个 tryLock()成功，都要在 finally 中调用 unLockMeAnd(res)❹，在该方法中测试并解除锁定。

2. 使用 ReadWriteLock

前面设计了线程安全的 ArrayList，如果有两个线程都想调用 get()与 size()方法，由于锁定的关系，一个线程只能等待另一个线程解除锁定，无法两个线程同时调用 get()与 size()方法。然而，这两个方法都只是读取对象状态，并没有变更对象状态，如果只是读取操作，可允许线程同时并行的话，那对读取效率将会有所改善。

ReadWriteLock 接口定义了读取锁定与写入锁定行为，可以使用 **readLock()**、**writeLock()** 方法返回 Lock 操作对象。**ReentrantReadWriteLock** 是 ReadWriteLock 接口的主要操作类，readLock() 方法会返回 **ReentrantReadWriteLock.ReadLock** 实例，writeLock() 方法会返回 **ReentrantReadWriteLock.WriteLock** 实例。

ReentrantReadWriteLock.ReadLock 操作了 Lock 接口，调用其 lock() 方法时，若没有任何 ReentrantReadWriteLock.WriteLock 调用过 lock() 方法，也就是没有任何写入锁定时，就可以取得读取锁定。

ReentrantReadWriteLock.WriteLock 操作了 Lock 接口，调用其 lock() 方法时，若没有任何 ReentrantReadWriteLock.ReadLock 或 ReentrantReadWriteLock. WriteLock 调用过 lock() 方法，也就是没有任何读取或写入锁定时，才可以取得写入锁定。

例如，使用 ReadWriteLock 改写前面的 ArrayList，改进读取效率：

Concurrency ArrayList2.java

```
package cc.openhome;

import java.util.Arrays;
import java.util.concurrent.locks.*;

public class ArrayList2<E> {
    private ReadWriteLock lock = new ReentrantReadWriteLock(); ←❶使用 ReadWriteLock
    private Object[] elems;
    private int next;

    public ArrayList2(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList2() {
        this(16);
    }

    public void add(E elem) {
        lock.writeLock().lock(); ←❷取得写入锁定
        try {
            if (next == elems.length) {
                elems = Arrays.copyOf(elems, elems.length * 2);
            }
            elems[next++] = elem;
        } finally {
            lock.writeLock().unlock(); ←❸解除写入锁定
        }
    }

    public E get(int index) {
        lock.readLock().lock();
        try {
            return (E) elems[index];
        } finally {
            lock.readLock().unlock();
        }
    }
```

```
    public int size() {
        lock.readLock().lock(); ◄── ❹ 取得读取锁定
        try {
            return next;
        } finally {
            lock.readLock().unlock(); ◄── ❺ 解除读取锁定
        }
    }
}
```

这次在 ArrayList 中使用 ReadWriteLock❶，若有线程调用 add()方法打算进行写入操作时，先取得写入锁定❷。这样若有其他线程打算再次取得写入锁定或取得读取锁定，都必须等待此次写入锁定解除。同样地，记得要在 finally 中解除写入锁定❸。

若有线程调用 get()方法打算进行读取操作时，先取得读取锁定❹。这样若有其他线程后续也可再取得读取锁定(例如有线程打算再调用 get()或 size()方法)，但若有线程打算取得写入锁定，必须等待所有读取锁定解除，记得要在 finally 中解除读取锁定❺。

这样设计之后，若线程都只是在调用 get()或 size()方法，则不会因等待锁定而进入阻断状态，可以提高读取效率。

3. 使用 StampedLock

ReadWriteLock 在没有任何读取或写入锁定时，才可以取得写入锁定，这可用于实现悲观读取(Pessimistic Reading)。如果线程进行读取时，经常可能有另一线程有写入需求，为了维持数据一致，ReadWriteLock 的读取锁定就可派上用场。

然而，如果读取线程很多，写入线程甚少的情况下，使用 ReadWriteLock 可能会使得写入线程遭受饥饿(Starvation)问题，也就是写入线程可能迟迟无法竞争到锁定，而一直处于等待状态。

JDK8 增加了 **StampedLock** 类，可支持乐观读取(Optimistic Reading)操作，也就是在读取线程很多写入线程甚少的情况下，你可以乐观地认为，写入与读取同时发生的机会甚少，因此乐观地使用完全的读取锁定，程序可以查看数据在被读取之后，是否遭到写入线程的变更，再采取后续的措施(重新读取变更后的数据，或者是抛出例外)。

假设之前的 ArrayList 范例会用于读取线程多而写入线程少的情况，而你想要操作乐观读取，如何使用 **StampedLock** 类来实现：

Concurrency ArrayList3.java

```
package cc.openhome;

import java.util.Arrays;
import java.util.concurrent.locks.*;

public class ArrayList3<E> {
    private StampedLock lock = new StampedLock(); ◄── ❶ 使用 StampedLock
    private Object[] elems;
    private int next;

    public ArrayList3(int capacity) {
        elems = new Object[capacity];
    }
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    public ArrayList3() {
        this(16);
    }

    public void add(E elem) {
        long stamp = lock.writeLock(); ←— ❷ 取得写入锁定
        try {
            if (next == elems.length) {
                elems = Arrays.copyOf(elems, elems.length * 2);
            }
            elems[next++] = elem;
        } finally {
            lock.unlockWrite(stamp); ←— ❸ 解除写入锁定
        }
    }

    public E get(int index) {
        long stamp = lock.tryOptimisticRead(); ←— ❹ 试着乐观读取锁定
        Object elem = elems[index];
        if (!lock.validate(stamp)) {  ←— ❺ 查询是否有排他的锁定
            stamp = lock.readLock();  ←— ❻ 真正地读取锁定
            try {
                elem = elems[index];
            } finally {
                lock.unlockRead(stamp); ←— ❼ 解除读取锁定
            }
        }
        return (E) elem;
    }

    public int size() {
        long stamp = lock.tryOptimisticRead();
        int size = next;
        if (!lock.validate(stamp)) {
            stamp = lock.readLock();
            try {
                size = next;
            } finally {
                lock.unlockRead(stamp);
            }
        }
        return size;
    }
}
```

范例中使用了 StampedLock❶，可以使用 writeLock()方法取得写入锁定，这会返回一个 long 整数代表锁定戳记(Stamp)❷，可用于使用解除锁定❸或通过 tryConvertToXXX()方法转换为其他锁定。

在范例 get()中示范了一种乐观读取的操作方式，**tryOptimisticRead()**不会真正执行读取锁定，而是返回锁定戳记❹，如果有其他排他性锁定的话，戳记会是 0，范例接着将数据暂读出至局部变量，**validate()**方法用来验证戳记是不是被其他排他性锁定取得了❺，如果是的话就返回 false，如果戳记是 0 也会返回 false。如果 if 验证出戳记被其他排他性锁定取得，重新使用 **readLock()**做真正的读取锁定❻，并在锁定时更新局部变量，然后解除读取锁定❼，如 if 验证条件不成立，只要直接返回局部变量的值。范例中的 size()方法也是类似的操作方式。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

注意 »» 在 validate() 之后发生写入而返回结果不一致是有可能的，如果你在意这样的不一致，应当采用完全的锁定。

提示 »» 以下文章 *Stamped Lock Idioms* 中比较了 synchronized、volatile、ReentrantLock、ReadWriteLock、StampedLock 等同步锁定机制：
http://javaspecialists.eu/archive/Issue215.html

4. 使用 Condition

Condition 接口用来搭配 Lock，最基本用法是达到 Object 的 wait()、notify()、notifyAll() 方法的作用。先来看看如何使用 Lock 与 Condition 改写 11.1.6 节中生产者、消费者范例中的 Clerk 类：

Concurrency Clerk.java

```
package cc.openhome;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Clerk {
    private int product = -1;
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition(); ←— ❶ 建立 Condition 对象

    public void setProduct(int product) throws InterruptedException {
        lock.lock();
        try {
            waitIfFull();
            this.product = product;
            System.out.printf("生产者设定 (%d)%n", this.product);
            condition.signal(); ←— ❷ 用 Condition 的 signal() 取代 Object 的 notify()
        } finally {
            lock.unlock();
        }
    }

    private void waitIfFull() throws InterruptedException {
        while (this.product != -1) {
            condition.await(); ←— ❸ 用 Condition 的 await() 取代 Object 的 wait()
        }
    }

    public int getProduct() throws InterruptedException {
        lock.lock();
        try {
            waitIfEmpty();
            int p = this.product;
            this.product = -1;
            System.out.printf("消费者取走 (%d)%n", p);
            condition.signal();
            return p;
        } finally {
            lock.unlock();
        }
    }
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    private void waitIfEmpty() throws InterruptedException {
        while (this.product == -1) {
            condition.await();
        }
    }
}
```

可以调用 Lock 的 **newCondition()** 取得 Condition 操作对象❶，调用 Condition 的 **await()** 将会使线程进入 Condition 的等待集合❸。要通知等待集合中的一个线程，则可以调用 **signal()** 方法❷；如果要通知所有等待集合中的线程，可以调用 **signalAll()** 方法。Condition 的 await()、signal()、signalAll()方法，可视为 Object 的 wait()、notify()、notifyAll()方法的对应。

事实上，11.1.6 节中的 Clerk 对象调用 wait()时，无论生产者还是消费者线程，都会至 Cleck 对象的等待集合，在多个生产者、消费者线程的情况下，等待集合中会有生产者与消费者线程，调用 notify()时，有可能通知到生产者线程，也有可能通知到消费者线程。如果现在消费者线程取走产品后，Clerk 没有产品了，而消费者最后 notify()时，实际又通知到消费者线程，那么只是让消费者线程再次执行到 wait()而重复进出等待集合。

事实上，一个 Condition 对象可代表有一个等待集合，可以重复调用 Lock 的 newCondition()，取得多个 Condition 实例，这代表了可以有多个等待集合。上面改写的 Clerk 类，因为使用了一个 Condition，所以实际上也只有一个等待集合，作用将类似 11.1.6 节中的 Clerk 类。如果可以有两个等待集合：一个是给生产者线程用，一个是给消费者线程用，生产者只通知消费者等待集合，消费者只通知生产者等待集合，效率会比较高。例如：

Concurrency Clerk2.java

```
package cc.openhome;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Clerk2 {
    private int product = -1;
    private Lock lock = new ReentrantLock();
    private Condition producerCond = lock.newCondition(); ←— ❶ 拥有生产者等待集合
    private Condition consumerCond = lock.newCondition(); ←— ❷ 拥有消费者等待集合

    public void setProduct(int product) throws InterruptedException {
        lock.lock();
        try {
            waitIfFull();
            this.product = product;
            System.out.printf("生产者设定 (%d)%n", this.product);
            consumerCond.signal(); ←— ❸ 通知消费者等待集合中的消费者线程
        } finally {
            lock.unlock();
        }
    }

    private void waitIfFull() throws InterruptedException {
        while (this.product != -1) {
            producerCond.await(); ←— ❹至生产者等待集合等待
        }
    }
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    public int getProduct() throws InterruptedException {
        lock.lock();
        try {
            waitIfEmpty();
            int p = this.product;
            this.product = -1;
            System.out.printf("消费者取走 (%d)%n", p);
            producerCond.signal(); ←— ❺通知生产者等待集合中的生产线程
            return p;
        } finally {
            lock.unlock();
        }
    }

    private void waitIfEmpty() throws InterruptedException {
        while (this.product == -1) {
            consumerCond.await(); ←— ❻至消费者等待集合等待
        }
    }
}
```

在这个范例中，分别为生产者线程与消费者线程建立了 Condition 对象，这表示可拥有两个等待集合，一个给生产者线程用❶，一个给消费者线程用❷。如果 Clerk 无法收东西了，那就请生产者线程至生产者等待集合中等待❹，而生产者线程设定产品后，会通知消费者等待集合中的线程❸。反之，消费者线程会至消费者等待集合中等待❻，若要通知，则通知生产者等待集合中的线程❺。

11.2.2 使用 Executor

Runnable 用来定义可执行流程与可使用数据，Thread 用来执行 Runnable。两者结合的基本做法正如前面介绍的，将 Runnable 指定给 Thread 创建之用，并调用 start() 开始执行。

Thread 的建立与系统资源有关，如何建立 Thread、是否重用 Thread、何时销毁 Thread、被指定的 Runnable 何时排定给 Thread 执行，这些都是复杂的议题。为此，从 JDK5 开始，定义了 **java.util.concurrent.Executor** 接口，目的是将 Runnable 的指定与实际执行分离。Executor 接口只定义了一个 **execute()** 方法：

```
package java.util.concurrent;
public interface Executor {
    void execute(Runnable command);
}
```

单看这个方法，你不会知道被指定的 Runnable 是如何被执行的。例如，可以将 11.1.3 节中的 Download 与 Download2 行为封装起来：

Concurrency Pages.java

```
package cc.openhome;

import java.net.URL;
import java.util.concurrent.*;
import java.io.*;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
public class Pages {
    private URL[] urls;
    private String[] fileNames;
    private Executor executor;

    public Pages(URL[] urls, String[] fileNames, Executor executor) {
        this.urls = urls;
        this.fileNames = fileNames;
        this.executor = executor;
    }

    public void download() {
        for (int i = 0; i < urls.length; i++) {
            URL url = urls[i];
            String fileName = fileNames[i];
            executor.execute(() -> {
                try {
                    dump(url.openStream(), new FileOutputStream(fileName));
                } catch (IOException ex) {
                    throw new RuntimeException(ex);
                }
            });
        }
    }

    private void dump(InputStream src, OutputStream dest)
                                            throws IOException {
        try (src; dest) {
            byte[] data = new byte[1024];
            int length;
            while ((length = src.read(data)) != -1) {
                dest.write(data, 0, length);
            }
        }
    }
}
```

单看这个 Pages 类，并不会知道实际上 Executor 如何执行给定的 Runnable 对象。如何执行，要看操作 Executor 的操作类如何定义，也许可以定义一个 DirectExecutor，单纯调用传入 execute()方法的 Runnable 对象的 run()方法。如：

Concurrency DirectExecutor.java

```
package cc.openhome;

import java.util.concurrent.Executor;

public class DirectExecutor implements Executor {
     public void execute(Runnable r) {
         r.run();
     }
}
```

如果这样使用 Pages 与 DirectExecutor：

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Concurrency Download.java

```
package cc.openhome;

import java.net.URL;
import java.io.*;

public class Download {
    public static void main(String[] args) throws Exception {
        URL[] urls = {
            new URL("http://openhome.cc/Gossip/Encoding/"),
            new URL("http://openhome.cc/Gossip/Scala/"),
            new URL("http://openhome.cc/Gossip/JavaScript/"),
            new URL("http://openhome.cc/Gossip/Python/")
        };

        String[] fileNames = {
            "Encoding.html",
            "Scala.html",
            "JavaScript.html",
            "Python.html"
        };

        new Pages(urls, fileNames, new DirectExecutor()).download();
    }
}
```

那就是只有主线程逐一执行指定的每个页面下载。如果定义一个 ThreadPerTaskExecutor：

Concurrency ThreadPerTaskExecutor.java

```
package cc.openhome;

import java.util.concurrent.Executor;

public class ThreadPerTaskExecutor implements Executor {
     public void execute(Runnable r) {
         new Thread(r).start();
     }
}
```

对于每个传入的 Runnable 对象，会建立一个 Thread 实例并 start() 执行。如果这样，使用 Pages 与 ThreadPerTaskExecutor：

Concurrency Download2.java

```
package cc.openhome;

import java.net.URL;
import java.io.*;

public class Download2 {
    public static void main(String[] args) throws Exception {
        ...这部分与 Download 相同，故略...
        new Pages(urls, fileNames, new ThreadPerTaskExecutor()).download();
    }
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

针对每个网页，会启动一个线程来进行下载。或许你会想到，若要下载的页面非常多，每次建立一个线程下载页面完后就丢弃该线程，过于浪费系统资源。也许你会想操作一个具有线程池(Thread Pool)的 Executor，建立可重复使用的线程，这是可行的。不过不用亲自操作，因为 Java SE API 中提供接口与操作类，可达到此类需求。在这之前，先来看看 Executor 的 API 架构，如图 11.6 所示。

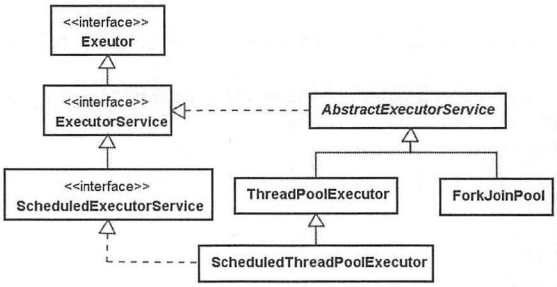


图 11.6 Executor API 架构图

1. 使用 ThreadPoolExecutor

在 Java SE API 中，像线程池这类服务的行为，实际上是定义在 Executor 的子接口 **java.util.concurrent.ExecutorService** 中。通用的 ExecutorService 由抽象类 AbstractExecutorService 操作，如果需要线程池的功能，则可以使用其子类 **java.util.concurrent.ThreadPoolExecutor**。根据不同的线程池需求，ThreadPoolExecutor 拥有数种不同构造函数可供使用。不过通常会使用 **java.util.concurrent.Executors** 的 **newCachedThreadPool()**、newFixedThreadPool()静态方法来创建 **ThreadPoolExecutor** 实例，程序看来较为清楚且方便。

Executors.newCachedThreadPool()返回的 ThreadPoolExecutor 实例，会在必要时建立线程，Runnable 可能执行在新建的线程，或重复利用的线程中，newFixedThreadPool()则可指定在池中建立固定数量的线程，这两个方法也都有接受 **java.util.concurrent.ThreadFactory** 的版本。可以在 ThreadFactory 的 **newThread()** 方法中，操作如何建立 Thread 实例。

例如，用 ThreadPoolExecutor 搭配前面的 Pages 使用：

Concurrency Download3.java

```
package cc.openhome;

import java.net.URL;

public class Download3 {
    public static void main(String[] args) throws Exception {
        ...这部分与 Download 相同，故略...

        ExecutorService executorService = Executors.newCachedThreadPool();
        new Pages(urls, fileNames, executorService).download();
        executorService.shutdown();
    }
}
```

ExecutorService 的 **shutdown()** 方法会在指定执行的 Runnable 都完成后，将 ExecutorService 关闭(在这里就是关闭 ThreadPoolExecutor)，另一个 **shutdownNow()** 方法，则可以立即关闭

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

ExecutorService，尚未被执行的 Runnable 对象会以 List<Runnable>返回。

ExecutorService 还定义了 **submit()**、**invokeAll()**、**invokeAny()**等方法，这些方法中出现了 **java.util.concurrent.Future**、**java.util.concurrent.Callable** 接口。先来看看这两个接口与相关 API 的架构，如图 11.7 所示。

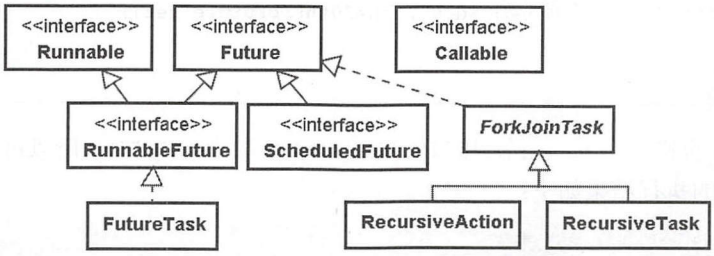


图 11.7 Future 与 Callable API 架构图

相信你应该有过这类对话的经验："老板，我要一份蚵仔煎，待会来拿！"这描述了 **Future** 定义的行为，就是让你在将来取得结果。你可以将想执行的工作交给 Future，Future 会使用另一线程来进行工作，你就可以先忙别的事去，过些时候，再调用 Future 的 **get()**取得结果，如果结果已经产生，get()会直接返回，否则会进入阻断直到结果返回。get()的另一版本则可以指定等待结果的时间，若指定的时间到但结果还没产生，就会抛出 **java.util.concurrent.TimeoutException**，也可以使用 Future 的 **isDone()**方法，看看结果是否产生。

Future 经常与 Callable 搭配使用，Callable 的作用与 Runnable 类似，可让你定义想要执行的流程。不过 Runnable 的 run()方法无法返回值，也无法抛出受检异常(Checked Exception)，然而 Callable 的 call()方法可以返回值，也可以抛出受检异常：

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

java.util.concurrent.FutureTask 是 Future 的操作类，创建时可传入 Callable 操作对象指定的执行的内容。来举个 Future 与 Callable 运用的实例：

Concurrency FutureCallableDemo.java

```
package cc.openhome;

import java.util.concurrent.*;
import static java.lang.System.*;

public class FutureCallableDemo {
    static long fibonacci(long n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void main(String[] args) throws Exception {
        FutureTask<Long> the30thFibFuture =
                new FutureTask<>(() -> fibonacci(30));

        out.println("老板，我要第 30 个费式数，待会来拿...");
```

```
        new Thread(the30thFibFuture).start();
        while(!the30thFibFuture.isDone()) {
            out.println("忙别的事去...");
        }

        out.printf("第 30 个费式数: %d%n", the30thFibFuture.get());
    }
}
```

由于 Future 也操作了 Runnable 接口(RunnableFutured 的父接口)，所以可以指定给 Thread 创建之用。范例的执行结果如下：

```
老板，我要第 30 个费式数，待会来拿...
忙别的事去...
第 30 个费式数: 832040
```

提示>>> 关于费式数，可以参考以下网址：
http://openhome.cc/Gossip/AlgorithmGossip/FibonacciNumber.htm

如果你的流程已定义在某个 Runnable 对象中，FutureTask 创建时也有接受 Runnable 的版本，并可指定一个对象在调用 get() 时返回(运算结束后)。

回头看看 ExecutorService 的 submit() 方法，它可以接受 Callable 对象，调用后返回的 Future 对象，就是让你在稍后可以取得运算结果。例如，改写以上的范例为使用 ExecutorService 的版本：

Concurrency FutureCallableDemo.java

```
package cc.openhome;

import java.util.concurrent.*;
import static java.lang.System.*;

public class FutureCallableDemo2 {
    static long fibonacci(long n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    public static void main(String[] args) throws Exception {
        ExecutorService service = Executors.newCachedThreadPool();

        out.println("老板，我要第 30 个费式数，待会来拿...");

        Future<Long> the30thFibFuture = service.submit(() -> fibonacci(30));
        while(!the30thFibFuture.isDone()) {
            out.println("忙别的事去...");
        }

        out.printf("第 30 个费式数: %d%n", the30thFibFuture.get());
    }
}
```

该范例的执行结果与前一个范例相同。如果有多个 Callable，可以先收集在 Collection 中，然后调用 ExecutorService 的 invokeAll()，这会以 List<Future<T>>返回与 Callable 相关联的 Future 对象。如果有多个 Callable，只要有一个执行完成就可以。先收集在 Collection 中，然后调用 ExecutorService 的 invokeAny()，只要 Collection 中有一个 Callable 完成，invokeAny() 就会返回该 Callable 的执行结果。

2. 使用 ScheduledThreadPoolExecutor

ScheduledExecutorService 为 ExecutorService 的子接口，可以让你进行工作排程；**schedule()** 方法用来排定 Runnable 或 Callable 实例延迟多久后执行一次，并返回 Future 子接口 **ScheduledFuture** 的实例；对于重复性的执行，可使用 **scheduleWithFixedDelay()** 与 **scheduleAtFixedRate()** 方法。

提示 »» 在还没有 ScheduledExecutorService 与相关操作类前(JDK5 开始提供)，标准 API 是由 java.util.Timer 与 java.util.TimerTask 提供排程功能(JDK1.3 就存在)，不过限制比较多。有兴趣的话，读者从网络上可以找到不少 Timer 与 TimerTask 的范例。

在一个线程只排定一个 Runnable 实例的情况下，scheduleWithFixedDelay() 方法可排定延迟多久首次执行 Runnable，执行完 Runnable 会排定延迟多久后再次执行。由于是以上次 Runnable 完成执行后的时间为准，所以运行时间就是排定时间。scheduleAtFixedRate() 可指定延迟多久首次执行 Runnable，同时依据指定周期排定每次执行 Runnable 的时间，如果上一次 Runnable 运行时间未超过指定周期，则运行时间就是排定时间；如果上次 Runnable 运行时间超过指定周期，上次 Runnable 执行完后，会立即执行下次 Runnable(运行时间就会晚于排定时间)。不管 scheduleWithFixedDelay() 还是 scheduleAtFixedRate()，上次排定的工作抛出异常，不会影响下次排程的进行。

ScheduledExecutorService 的操作类 **ScheduledThreadPoolExecutor** 为 ThreadPoolExecutor 的子类，具有线程池与排程功能。可以使用 Executors 的 **newScheduledThreadPool()** 方法指定返回内建多少个线程的 ScheduledThreadPoolExecutor；**newSingleThreadScheduledExecutor()** 则可使用单个线程执行排定的工作。

以下范例示范 newSingleThreadScheduledExecutor() 返回的 ScheduledExecutorService，在排定一个 Runnable 的情况下，scheduleWithFixedDelay() 执行的时间点。

Concurrency ScheduledExecutorServiceDemo.java

```
package cc.openhome;

import java.util.concurrent.*;

public class ScheduledExecutorServiceDemo {
    public static void main(String[] args) {
        ScheduledExecutorService service
                = Executors.newSingleThreadScheduledExecutor();

        service.scheduleWithFixedDelay(
                () -> {
                    System.out.println(new java.util.Date());
                    try {
                        Thread.sleep(2000); // 假设这个工作会进行两秒
                    } catch (InterruptedException ex) {
```


严禁网络传播本PDF，违者责任自负！

```
                    throw new RuntimeException(ex);
                }
            }, 2000, 1000, TimeUnit.MILLISECONDS);
    }
}
```

java.util.Date 创建时，会取得当时系统时间。每次工作会执行 2 秒，而后延迟 1 秒，所以看到的时间显示总共是 3 秒为一个间隔：

```
Wed Apr 16 11:46:09 CST 2018
Wed Apr 16 11:46:12 CST 2018
Wed Apr 16 11:46:15 CST 2018
Wed Apr 16 11:46:18 CST 2018
Wed Apr 16 11:46:21 CST 2018
Wed Apr 16 11:46:24 CST 2018
Wed Apr 16 11:46:27 CST 2018
```

如果把以上范例的 scheduleWithFixedDelay()换为 scheduleAtFixedRate()，每次排定的执行周期虽然为 1 秒，但由于每次工作执行实际上为 2 秒，会超过排定周期，所以上一次执行完工作后，会立即执行下一次工作，结果就是时间显示为 2 秒一个间隔：

```
Wed Apr 16 11:47:51 CST 2018
Wed Apr 16 11:47:53 CST 2018
Wed Apr 16 11:47:55 CST 2018
Wed Apr 16 11:47:57 CST 2018
Wed Apr 16 11:47:59 CST 2018
Wed Apr 16 11:48:01 CST 2018
```

如果再把 Thread.sleep(2000)改为 Thread.sleep(500)，由于每次工作执行不会超过排定周期，所以时间显示会 1 秒一个间隔：

```
Wed Apr 16 11:48:34 CST 2018
Wed Apr 16 11:48:35 CST 2018
Wed Apr 16 11:48:36 CST 2018
Wed Apr 16 11:48:37 CST 2018
Wed Apr 16 11:48:38 CST 2018
Wed Apr 16 11:48:39 CST 2018
```

提示»» 这三个范例如果排定的 Runnable 超过两个以上会如何呢？如果再改用 Executors 的 newScheduledThreadPool()方法，建立内建多个线程的线程池，执行结果又会如何呢？为了不影响多个排定工作的运行时间，可建立内建足够数量的线程。

3. 使用 ForkJoinPool

图 11.7 中，Future 的另一操作类 **java.util.concurrent.ForkJoinTask** 及其子类，与图 11.6 中 ExecutorService 的另一操作类 **java.util.concurrent.ForkJoinPool** 有关，它们都是 JDK7 中新增的 API，主要目的是在解决分而治之(Divide and Conquer)的问题。

所谓分而治之的问题，是指这些问题的解决可以分解为性质相同的子问题，子问题还可以再分解为更小的子问题，将性质相同的子问题解决并收集运算结果(如果必要的话)，整体问题也就解决了，如图 11.8 所示。

严禁网络传播本PDF，违者责任自负！

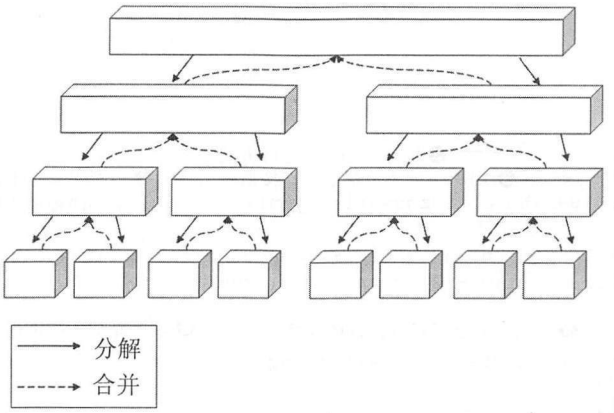


图 11.8　分而治之的概念

费式数列的解决就是一个例子。观察费式数演算伪码就可以了解：

```
Procedure FIB(N) [
    IF (N < 0)
        PRINT ("输入错误");

    IF (N = 0 OR N = 1)
        RETURN (N);
    ELSE
        RETURN ( FIB(N-1) + FIB(N-2) );
]
```

第 N 个费式数的求解，可以分解为第 N-1 个费式数与第 N-2 个费式数求解两个子任务，之后再将两数总和相加即可求得。至于第 N-1 个费式数，可再分解为第 N-2 与 N-3 个费式数求解。第 N-2 个费式数，则可再分解为第 N-3 与 N-4 个费式数求解。

分而治之可以结合并行，以费式数求解为例，若以循序方式解决第 N 个费式数，必须先求得第 N-1 个费式数，再求得第 N-2 个费式数，如果环境中允许使用两个以上的处理流程，第 N-1 个费式数与第 N-2 个费式数以及各自子任务同时并行，那么解决问题的速度将会加快。

在分而治之需要结合并行的情况下，可以使用 `ForkJoinTask`，其操作了 `Future` 接口，可让你在未来取得耗时工作的执行结果。如果有个 `ForkJoinTask` 在 `ForkJoinPool` 的管理时，执行了 **`fork()`** 方法，则会以另一个线程来执行它；如果想取得 `ForkJoinTask` 的执行结果，可以调用 **`join()`** 方法；如果执行结果尚未产生，则会阻断至执行结果返回。

使用 `ForkJoinTask` 的子类 `RecursiveTask`，用于执行后会返回结果的时候，`RecursiveTask` 是个抽象类，使用时必须继承它，并操作 `compute()` 方法。如果分而治之的过程中，子任务不需返回值，可以继承 `RecursiveAction` 并操作 `compute()` 方法。例如，下面是使用 `RecursiveTask` 与 `ForkJoinPool` 解决费式数的示范。

Concurrency FibonacciForkJoin.java

```
package cc.openhome;

import java.util.concurrent.*;

class Fibonacci extends RecursiveTask<Long> {  ←❶ 继承 RecursiveTask
    final long n;
```


严禁网络传播本PDF，违者责任自负！

```
    Fibonacci(long n) {
        this.n = n;
    }

    @Override
    public Long compute() {  ←❷操作 compute()方法
        if (n <= 20) {  ←❸20 以下就不分解了，直接循序运算
            return solveFibonacciSequentially(n);
        }                                          ❹分解出 n-1 子任务、请 ForkJoinPool
        ForkJoinTask<Long> subTask = new Fibonacci(n - 1).fork();   分配线程来执行这个子任务
        return new Fibonacci(n - 2).compute() + subTask.join();
    }
                          ❺分解出 n-2 子任务并直接运算        ❻取得此子任务执行结果
    static long solveFibonacciSequentially(long n) {
        if (n <= 1) {
            return n;
        }
        return solveFibonacciSequentially(n - 1)
                + solveFibonacciSequentially(n - 2);
    }
}

public class FibonacciForkJoin {

    public static void main(String[] args) {
        Fibonacci fibonacci = new Fibonacci(45);
        ForkJoinPool pool = new ForkJoinPool();
        System.out.println(pool.invoke(fibonacci));  ←❼开始分而治之
    }
}
```

在继承 RecursiveTask 之后❶，主要是将子任务的分解与求解过程撰写于 compute()方法之中❷，为了避免分解出过多的子任务，造成不必要的负担，n 小于 20 就不分解出子任务❸，直接用循序求解，在这里 n 设定为 20 只是纯粹作为示范，你必须决定子任务的最佳数量。

其中进行了 n-1 与 n-2 子任务的分解，n-1 子任务调用了 fork()方法❹，ForkJoinPool 会分配线程来执行其 compute()方法，程序中直接调用 n-2 任务的 compute()❺，在取得 n-2 费式数的值后，幸运的话，n-1 的值也许就运算好了，此时直接调用 join()来取得运算结果(如果还没，就会待到执行结果返回)❻。ForkJoinPool 的 invoke()方法，接受 ForkJoinTask 实例❼，如果是 RecursiveTask，将会调用其 compute()开始执行运算，所有 ForkJoinTask 实例的 compute()方法执行完毕，ForkJoinPool 就会关闭。

在多处理器的环境中，每个处理器若分配一个线程，每个线程分别执行一个子任务，由于各个处理器指令周期、数据处理量等可能不同，会造成各线程子任务完成的时间不一，如果某个线程完成了子任务，该核心运算资源也就闲置了下来。

ForkJoinPool 与其他的 ExecutorService 操作不同的地方在于，它实现了工作窃取(Work-stealing)演算，其建立的线程如果完成手边任务，会尝试寻找并执行其他任务建立的子任务，让线程保持忙碌状态，有效利用处理器的能力。默认 ForkJoinPool 会依处理器数量来建立线程，你可以通过 Runtime.getRuntime().availableProcessors()得知可用的处理器数量，在使用 ForkJoin 框架时，建立超过处理器数量的线程，基本上对效率不会有更多的帮助。为了让处理器保持忙碌状态，ForkJoin 框架适用于计算密集式的任务，不适用于容易造成线程阻断的场合，例如 I/O 密集式的任务。

严禁网络传播本PDF，违者责任自负！

11.2.3 并行 Collection 简介

在 java.util.concurrent 包中，提供了一些支持并行操作的 Collection 子接口与操作类。下面简介一些常用的接口与类。

如果使用第 9 章介绍过的 List 操作，由于它们都是非线程安全类，为了要在使用迭代器时，不受另一线程写入操作的影响，必须做类似以下的动作：

```
List list = new ArrayList();
...
synchornized(list) {
    Iterator iterator = list.iterator();
    while(iterator.hasNext()) {
        ...
        }
}
```

使用 Collections.synchronizedList()并非就不用如下的操作，它返回的实例保证使 List 操作时的线程安全，而非保证返回的 Iterator 操作时的线程安全。所以使用迭代器操作时，仍得类似以下操作：

```
List list = Collections.synchronizedList(new ArrayList());
...
synchornized(list) {
    Iterator iterator = list.iterator();
    while(iterator.hasNext()) {
        ...
    }
}
```

提示 »» 别忘了增强式 for 循环语法使用在 Collection 时，底层也是使用迭代器，所以在多线程存取下，也得类似以下写法：

```
List list = Collections.synchronizedList(new ArrayList());
...
synchronized(list) {
    for(Object o : ) {
...
    }
}
```

CopyOnWriteArrayList 操作了 List 接口，顾名思义，这个类的实例在写入操作时(如 add()、set()等)，内部会建立新数组，并复制原有数组索引的参考，然后在新数组上进行写入操作，写入完成后，再将内部原参考旧数组的变量参考至新数组。

对写入而言，这是个很耗资源的设计，然而在使用迭代器时，写入不会影响迭代器已参考的对象。对于一个很少进行写入操作而使用迭代器频繁的情境下，可以使用 CopyOnWriteArrayList 提高迭代器操作的效率。

CopyOnWriteArraySet 操作了 Set 接口，内部使用 CopyOnWriteArrayList 来完成 Set 的各种操作，因此一些特性与 CopyOnWriteArrayList 是相同的。例如，写入操作时会建立新数组复制所有索引参考而较耗成本，但在使用迭代器操作时会有较高的效率，适用于一个很少进行写入操作而使用迭代器频繁的情境。

BlockingQueue 是 Queue 的子接口，新定义了 **put()**与 **take()**等方法，线程若调用 put()方

严禁网络传播本PDF，违者责任自负！

法，在队列已满的情况下会被阻断，线程若调用 take() 方法，在队列为空的情况下会被阻断。有了这个特性，11.1.6 节中生产者与消费者的范例，就不用自行设计 Clerk 类，而可以直接使用 BlockingQueue。例如，Producer 可以改为以下：

Concurrency Producer3.java

```
package cc.openhome;

import java.util.concurrent.BlockingQueue;

public class Producer3 implements Runnable {
    private BlockingQueue<Integer> productQueue;

    public Producer3(BlockingQueue<Integer> productQueue) {
        this.productQueue = productQueue;
    }

    public void run() {
        System.out.println("生产者开始生产整数......");
        for(int product = 1; product <= 10; product++) {
            try {
                productQueue.put(product);
                System.out.printf("生产者提供整数 (%d)%n", product);
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

Consumer3 可以改为以下：

Concurrency Consumer3.java

```
package cc.openhome;

import java.util.concurrent.BlockingQueue;

public class Consumer3 implements Runnable {
    private BlockingQueue<Integer> productQueue;

    public Consumer3(BlockingQueue<Integer> productQueue) {
        this.productQueue = productQueue;
    }

    public void run() {
        System.out.println("消费者开始消耗整数......");
        for(int i = 1; i <= 10; i++) {
            try {
                int product = productQueue.take();
                System.out.printf("消费者消费整数 (%d)%n", product);
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

严禁网络传播本PDF，违者责任自负！

可以使用 BlockingQueue 的操作 **ArrayBlockingQueue** 类，这样就不用处理麻烦的 wait()、notify()等流程。例如：

Concurrency ProducerConsumerDemo3.java

```
package cc.openhome;

import java.util.concurrent.*;

public class ProducerConsumerDemo3 {
    public static void main(String[] args) {
        BlockingQueue queue = new ArrayBlockingQueue(1);      // 容量为 1
        new Thread(new Producer3(queue)).start();
        new Thread(new Consumer3(queue)).start();
    }
}
```

BlockingQueue 还有其他操作，以及 BlockingQueue 子接口和相关操作类，可以查看 API 文件来得知操作原理与相关使用。

ConcurrentMap 是 Map 的子接口，其定义了 **putIfAbsent()**、**remove()**与 **replace()**等方法，这些方法都是原子(Atomic)操作。putIfAbsent()在键对象不存在 ConcurrentMap 中时，才可置入键/值对象，否则返回键对应的值对象，也就是相当于你自行在 synchronized 中进行以下动作：

```
if (!map.containsKey(key)) {
    return map.put(key, value);
} else {
    return map.get(key);
}
```

remove()只有在键对象存在，且对应的值对象等于指定的值对象，才将键/值对象移除，也就是相当于你自行在 synchronized 中进行以下动作：

```
if (map.containsKey(key) && map.get(key).equals(value)) {
    map.remove(key);
    return true;
} else return false;
```

replace()有两个版本，其中一个版本是只有在键对象存在，且对应的值对象等于指定的值对象，才将值对象置换，也就是相当于你自行在 synchronized 中进行以下动作：

```
if (map.containsKey(key) && map.get(key).equals(oldValue)) {
   map.put(key, newValue);
   return true;
} else return false;
```

另外一个版本是在键对象存在时，将值对象置换，也就是相当于你自行在 synchronized 中进行以下动作：

```
if (map.containsKey(key)) {
    return map.put(key, value);
} else return null;
```

ConcurrentHashMap 是 ConcurrentMap 的操作类，ConcurrentNavigableMap 是 ConcurrentMap 子接口，其操作类为 **ConcurrentSkipListMap**，可视为支持并行操作的 TreeMap 版本。可以查看

严禁网络传播本PDF，违者责任自负！

API 文件来得知相关使用方式与操作原理。

提示>>> ConcurrentHashMap 中 putIfAbsent()、remove()与 replace()等方法，在 JDK8 时被定义在 Map 接口为默认方法，可以参考文章《Map 便利的预设方法》：http://openhome.cc/Gossip/CodeData/JDK8/Map.html

11.3 重点复习

要让目前流程暂停指定时间，可以使用 java.lang.Thread 的静态 sleep()方法，指定的单位是毫秒，调用这个方法必须处理 java.lang.InterruptedException。

如果想在 main()以外独立设计流程，可以撰写类操作 java.lang.Runnable 接口，流程的进入点是操作在 run()方法中。从 main()开始的流程会由主线程执行，可以创建 Thread 实例来执行 Runnable 实例定义的 run()方法，要启动线程执行指定流程，必须调用 Thread 实例的 start()方法。

除了将流程定义在 Runnable 的 run()方法中之外，另一个撰写多线程程序的方式，就是继承 Thread 类，重新定义 run()方法。操作 Runnable 接口的好处就是较有弹性，你的类还有机会继承其他类。若继承了 Thread，那该类就是一种 Thread，通常是为了直接利用 Thread 中定义的一些方法，才会继承 Thread 来操作。

如果主线程中启动了额外线程，默认会等待被启动的所有线程都执行完 run()方法才中止 JVM。如果一个 Thread 被标示为 Daemon 线程，在所有的非 Daemon 线程都结束时，JVM 自动就会终止。

在调用 Thread 实例 start()方法后，基本状态为可执行(Runnable)、被阻断(Blocked)、执行中(Running)。

线程有其优先权，可使用 Thread 的 setPriority()方法设定优先权，可设定值为 1(Thread.MIN_PRIORITY)到 10(Thread.MAX_PRIORITY)，默认是 5(Thread.NORM_PRIORITY)，超出 1 到 10 外的设定值会抛出 IllegalArgumentException。数字越大优先权越高，排班器越优先排入 CPU，如果优先权相同，则输流执行(Round-robin)。

当某线程进入 Blocked 时，将另一线程排入 CPU 执行(成为 Running 状态)，避免 CPU 空闲下来，经常是改进效能的方式之一。

线程因输入/输出进入 Blocked 状态后，在阻断情况结束后，会回到 Runnable 状态，等待排班器排入执行(Running 状态)。一个进入 Blocked 状态的线程，可以由另一个线程调用该线程的 interrupt()方法，让它离开 Blocked 状态。

如果 A 线程正在运行，流程中允许 B 线程加入，等到 B 线程执行完毕后再继续 A 线程流程，则可以使用 join()方法完成这个需求。

线程完成 run()方法后，就会进入 Dead，进入 Dead(或已经调用过 start()方法)的线程不可以再次调用 start()方法，否则会抛出 IllegalThreadStateException。

Thread 类上定义有 stop()方法，不过被标示为 Deprecated，表示过去确实定义过，后来因为会引发某些问题，为了确保向前兼容性，这些 API 没有直接剔除，但不建议新撰写的程序再使用它。

如果要停止线程，最好自行操作，让线程跑完应有的流程，而非调用 Thread 的 stop()方法。不仅停止线程必须自行根据条件操作，线程的暂停、重启，也必须视需求操作，而不是直接调

严禁网络传播本PDF，违者责任自负！

用 suspend()、resume()等方法。

每个线程都属于某个线程群组。若在 main()主流程中产生一个线程，该线程会属于 main 线程群组。每个线程产生时，都会归入某个线程群组，这视线程是在哪个群组中产生，如果没有指定，则归入产生该子线程的线程群组。也可以自行指定线程群组，线程一旦归入某个群组，就无法更换群组。

线程存取同一对象相同资源时可能引发竞速情况，这样的类为不具备线程安全的类。

每个对象都会有个内部锁定，或称为监控锁定。被标示为 synchronized 的区块将会被监控，任何线程要执行 synchronized 区块都必须先取得指定的对象锁定。

如果在方法上标示 synchronized，则执行方法必须取得该实例的锁定。synchronized 不仅可以声明在方法上，也可以描述句方式使用，在线程要执行 synchronized 区块时，必须取得括号中指定的对象锁定。

Java 的 synchronized 提供的是可重入同步，也就是线程取得某对象锁定后，若执行过程中又要执行 synchronized，尝试取得锁定的对象又是同一个，则可以直接执行。

执行 synchronized 范围的程序代码期间，若调用锁定对象的 wait()方法，线程会释放对象锁定，并进入对象等待集合而处于阻断状态，其他线程可以竞争对象锁定，取得锁定的线程可以执行 synchronized 范围的程序代码。

放在等待集合的线程不会参与 CPU 排班，wait()可以指定等待时间，时间到之后线程会再次加入排班，如果指定时间为 0 或不指定，则线程会持续等待，直到被中断(调用 interrupt())或是告知(notify())可以参与排班。

被竞争锁定的对象调用 notify()时，会从对象等待集合中随机通知一个线程加入排班，再次执行 synchronized 前，被通知的线程会与其他线程共同竞争对象锁定；如果调用 notifyAll()，所有等待集合中的线程都会被通知参与排班，这些线程会与其他线程共同竞争对象锁定。

11.4 课后练习

如果有个线程池可以分配线程来执行 Request 操作对象的 execute()方法，执行完后该线程类必须能重复使用，该线程类如何设计呢？假设 Request 接口定义如下：

```
public interface Request {
    void execute();
}
```

非卖品，仅供非商业用途或交流学习使用

Chapter 12

Lambda

学习目标

- 认识 Lambda 语法
- 运用方法参考
- 了解接口默认方法
- 使用 Functional 与 Stream API
- Lambda、平行化与非同步处理

非卖品，仅供非商业用途或交流学习使用

12.1 认识 Lambda 语法

在第 9 章你曾经看到了 Lambda 语法的简介，并试着在那之后的章节范例中，可以替换匿名类的场合运用 Lambda 语法，以取得语法的简洁，增加程序代码的表达性。不过，那并不是 Lambda 项目的全部，在这一章，将会来完整认识 Lambda，了解如何使用 Lambda。

12.1.1 Lambda 语法概览

在 9.1.6 节曾经简介过 Lambda 语法，不过，为了整体介绍的完整性，本节将重新予以介绍。类似地，先来看个匿名类的应用场合，举例而言，如果你打算将用户名称按长度进行排序，在 JDK8 出现前，可以如下撰写程序：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
});
```

Arrays 的 sort() 方法可以用来排序，只不过，你得告诉它两个元素比较时顺序是什么，sort() 规定得操作 java.util.Comparator 来说明这件事，然而那个匿名类的语法有些冗长。先别急着使用 Lambda 语法，如果想稍微改变一下 Arrays.sort() 该行的可读性，在 JDK8 出现前，可以如下撰写：

```
Comparator<String> byLength = new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
};

String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, byLength);
```

通过变量 byLength，确实是可以让排序的意图清楚许多，只是操作 Comparator 时的匿名类依旧冗长，有太多重复的信息，如果使用 JDK8 的话，可以使用 Lambda 特性去除重复的信息。

例如，声明 byLength 时已经写了一次 Comparator<String>，为什么操作匿名类时又得写一次 Comparator<String>？使用 JDK8 的 Lambda 表达式的话，可以写为：

```
Comparator<String> byLength =
     (String name1, String name2) -> name1.length() - name2.length();
```

重复的 Comparator<String>信息从等号右边去除了，因为原本的匿名类只有一个方法要操作，因此在使用 Lambda 表达式时，实际上从等号左边的 Comparator<String>声明就可以知道，Lambda 表达式实际上是要操作 Comparator<String>的 compare() 方法。仔细看看，还有重复的信息，既然声明变量时使用了 Comparator<String>，为什么 Lambda 表达式的参数上又得声明一次 String？实际上确实不用，因为编译程序可以从 byLength 变量的声明类型，推断 name1 与 name2 的类型，因此可以再简化为：

非卖品，仅供非商业用途或交流学习使用

```
Comparator<String> byLength = (name1, name2) -> name1.length() - name2.length();
```

等号右边的表达式够简短了，不如将它直接放到 Arrays 的 sort() 方法中：

Lambda LambdaDemo.java

```
package cc.openhome;

import java.util.Arrays;

public class LambdaDemo {
    public static void main(String[] args) {
        String[] names = {"Justin", "caterpillar", "Bush"};
        Arrays.sort(names, (name1, name2) -> name1.length() - name2.length());
        System.out.println(Arrays.toString(names));
    }
}
```

因为编译程序可以从 names 推断，sort() 方法的第二个参数类型实际上就是 Comparator<String>，因而 name1 与 name2 还是不用声明类型；跟一开始的匿名类写法相比较，这样的程序代码确实是简洁许多，那么，Lambda 只是匿名类的语法蜜糖吗？不！还有许多细节会在后续介绍，现在还是先集中去除重复性与改善可读性。

提示 »» 本书只从 Java 的角度来介绍 Lambda，如果对 Lambda 的前世今生有兴趣，可以参考 *Java Lambda Tutorial*：http://openhome.cc/Gossip/CodeData/index.html#JavaLambdaTutorial。

如果有按字符串长度排序的需求，那你会怎么做？如果要求同一个方法内完成，那么就用 byName 局部变量吧！如果是类中多个方法间要共享，那就用一个 byName 的数据成员吧！因为 byName 要参考的实例没有状态问题，因而声明为 static 比较适合。如果要在多个类之间共享，那么就设定为 public static。例如：

Lambda StringOrder.java

```
package cc.openhome;

public class StringOrder {
    public static int byLength(String s1, String s2) {
        return s1.length() - s2.length();
    }

    public static int byLexicography(String s1, String s2) {
        return s1.compareTo(s2);
    }

    public static int byLexicographyIgnoreCase(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
}
```

将一些字符串排序时可能的方式都定义出来了，原本的按名称长度排序就可以改写为：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, (name1, name2) -> StringOrder.byLength(name1, name2));
```

也许你发现了，除了方法名称之外，byLength 方法的签署与 Comparator 的 compare() 方法相同，你只是在 Lambda 表达式中将参数 s1 与 s2 传给 byLength 方法，这不是重复那什么叫重

非卖品，仅供非商业用途或交流学习使用

复？可以直接重用 byLength 方法的操作不是更好吗？JDK8 之后提供了方法参考(Method Reference)的特性，可以达到这个目的：

Lambda StringOrderDemo.java

```
package cc.openhome;

import java.util.Arrays;

public class StringOrderDemo {
    public static void main(String[] args) {
        String[] names = {"Justin", "caterpillar", "Bush"};
        Arrays.sort(names, StringOrder::byLength);
        System.out.println(Arrays.toString(names));
    }
}
```

在 Java 中引入 Lambda 的同时，与现有 API 维持兼容性是主要考虑之一。方法参考的特性，在重用现有 API 上扮演了重要角色。重用现有方法操作，可避免到处写下 Lambda 表达式。上面的例子是运用了方法参考中的一种形式，也就是参考了现有的 static 方法。

现在来看看另一个需求，如果想按字典顺序排序名称呢？因为已经定义了 StringOrder，所以可以这么撰写：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, StringOrder::byLexicography);
```

嗯！？仔细看看，StringOrder 的 byLexicography() 方法操作中，只不过是调用 String 的 compareTo() 方法，也就是将第一个参数 s1 作为 compareTo() 的接受者，第二个参数 s2 作 compareTo() 方法的参数，在这种情况下，可以直接参考 String 类的 compareTo 方法，例如：

Lambda StringDemo.java

```
package cc.openhome;

import java.util.Arrays;

public class StringDemo {
    public static void main(String[] args) {
        String[] names = {"Justin", "caterpillar", "Bush"};
        Arrays.sort(names, String::compareTo);
        System.out.println(Arrays.toString(names));
    }
}
```

类似地，想对名称按照字典顺序排序，但忽略大小写差异，也不用再通过 StringOrder 的 static 方法了，只需要直接参考 String 的 compareToIgnoreCase() 方法：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, String::compareToIgnoreCase);
```

可以观察到，方法参考不仅避免了重复撰写 Lambda 表达式，也可以让程序代码更为清楚。这边只是初尝一下 Lambda 的甜头，关于 Lambda 还有更多细节，后续会再来逐一介绍。

非卖品，仅供非商业用途或交流学习使用

12.1.2 Lambda表达式与函数接口

看过了Lambda的几个应用范例，接下来得了解一些细节了。首先，你得知道以下的程序代码：

```
Comparator<String> byLength =
     (String name1, String name2) -> name1.length() - name2.length();
```

可以拆开为两部分，等号右边是Lambda表达式(Expression)，等号左边是作为Lambda表达式的目标类型(Target Type)。先来看看等号右边的Lambda表达式：

```
(String name1, String name2) -> name1.length() - name2.length()
```

这个Lambda表达式表示接受两个参数name1、name2，参数都是String类型，目前->右边定义了会返回结果的单一表达式，如果运算比较复杂，必须使用多行描述，可以加入{}定义描述区块，如果有返回值，必须加上return，例如：

```
(String name1, String name2) -> {
    String name1 = name1.trim();
    String name2 = name2.trim();
    ...
    return name1.length() - name2.length();
}
```

区块可以由数个描述语句组成，不过基本上不建议如此使用。在运用Lambda表达式时，尽量使用简单的表达式，如果操作比较复杂，可以考虑方法参考等其他方式。

Lambda表达式中，即使不接受任何参数，也必须写下括号。例如：

```
() -> "Justin"                // 不接受参数，返回字符串
() -> System.out.println()    // 不接受参数，没有返回值
```

在只有Lambda表达式的情况下，参数的类型必须写出来，如果有目标类型的话，在编译程序可推断出类型的情况下，就可以不写出Lambda表达式的参数类型。例如以下范例可以从Comparator<String>中推断出name1与name2的类型，实际上是String，因而就不用写出参数类型：

```
Comparator<String> byLength = (name1, name2) -> name1.length() - name2.length();
```

Lambda表达式本身是中性的，不代表任何类型的实例，同样的Lambda表达式可用来表示不同目标类型的对象操作。举例而言，(name1, name2) -> name1.length() - name2.length()在上面的范例中，用来表示Comparator<String>的操作，如果你定义了一个接口：

```
public interface Func<P, R> {
    R apply(P p1, P p2);
}
```

那么同样是(name1, name2) -> name1.length() - name2.length()这个表达式，在以下的程序代码中：

```
Func<String, Integer> func = (name1, name2) -> name1.length() - name2.length();
```

就是用来表示目标类型为Func<String, Integer>的操作，这个例子也示范了如何定义Lambda表达式的目标类型，Lambda并没有导入新类型来作为Lambda表达式的类型，而是就现有的interface语法来定义函数接口(Functional Interface)，作为Lambda表达式的目标类型。函数接口就是接口，但要求仅具单一抽象方法，许多现存的接口都是这种接口，如标准API中的Runnable、Callable、Comparator等，都只定义了一个方法。

非卖品，仅供非商业用途或交流学习使用

```
    public interface Runnable {
        void run();
    }

    public interface Callable<V> {
        V call() throws Exception;
    }

public interface Comparator<T> {
        int compare(T o1, T o2);
    }
```

在JDK8出现前，可以使用匿名类来操作这类接口，匿名类不是不好，只不过有其应用的场合。然而在许多时候，特别是接口只有一个方法要操作时，你会只关心参数及操作本身，不理会类与方法名称，例如12.1.1节中以匿名类操作的例子：

```
Arrays.sort(names, new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.length() - name2.length();
    }
});
```

实际上，就这个代码段而言，你关心的只是怎么比较两个元素。在这类情况下，使用Lambda表达式会让你能专注在程序代码的意图：

```
Arrays.sort(names, (name1, name2) -> name1.length() - name2.length());
```

所以，Lambda表达式只关心方法签署上的参数与返回定义，但忽略方法名称。如果函数接口上定义的方法只接受一个参数，例如：

```
public interface Func {
    public void apply(String s);
}
```

你在撰写Lambda表达式时，若编译程序可推断出类型，本来可以写为：

```
Func func = (s) -> out.println(s);
```

这时括号就是多余的了，可以省略写为：

```
Func func = s -> out.println(s);
```

函数接口是仅具单一抽象方法的接口，不过有时会难以直接分辨接口是否为函数接口，稍后就会看到，因为JDK8对interface语法做了改进，允许有默认方法(Default Method)。而接口可能继承其他接口、重新定义了某些方法等，这些都会使得确认接口是否为函数接口更为困难。有个新标注**@FunctionalInterface**在JDK8中被引入，它可以这么使用：

```
@FunctionalInterface
public interface Func<P, R> {
    R apply(P p);
}
```

如果接口使用了@FunctinalInterface来标注，而本身并非函数接口的话，就会引发编译错误。例如以下这个接口：

```
@FunctionalInterface
public interface Function<P, R> {
    R call(P p);
```


非卖品，仅供非商业用途或交流学习使用

```
    R call(P p1, P p2);
}
```

编译程序会对此接口产生以下编译错误：

```
@FunctionalInterface
^
Function is not a functional interface
  multiple non-overriding abstract methods found in interface Function
1 error
```

12.1.3 Lambda 遇上 this 与 final

Lambda 表达式并不是匿名类的语法蜜糖，如果将它当作语法蜜糖，在处理 this 实际参考对象时，就会觉得困惑。先来看一下接下来的程序，其中使用了匿名类，先想想看结果会如何显示？

Lambda ThisDemo.java

```
package cc.openhome;

import static java.lang.System.out;

class Hello {
    Runnable r1 = new Runnable() {
        public void run() {
            out.println(this);
        }
    };

    Runnable r2 = new Runnable() {
        public void run() {
            out.println(toString());
        }
    };

    public String toString() {
        return "Hello, world!";
    }
}

public class ThisDemo {
    public static void main(String[] args) {
        Hello hello = new Hello();
        hello.r1.run();
        hello.r2.run();
    }
}
```

你认为执行结果会显示“Hello, World!”吗？看来并不是：

```
cc.openhome.Hello$1@15db9742
cc.openhome.Hello$2@6d06d69c
```

这是因为在这个范例中，this 的参考对象以及 toString()(也就是 this.toString())的接受者，实际上都是匿名类建立的实例，也就是 Runnable 实例，由于你没有定义 Runnable 的 toString()，因而显示结果是 Object 默认的 toString()返回字符串。再来看看接下来的程序，它会显示什么？

非卖品，仅供非商业用途或交流学习使用

Lambda ThisDemo2.java

```
package cc.openhome;

import static java.lang.System.out;

class Hello2 {
    Runnable r1 = () -> out.println(this);
    Runnable r2 = () -> out.println(toString());

    public String toString() {
        return "Hello, world!";
    }
}

public class ThisDemo2 {
    public static void main(String[] args) {
        Hello2 hello = new Hello2();
        hello.r1.run();
        hello.r2.run();
    }
}
```

如果 Lambda 表达式只是匿名类的语法蜜糖，那么结果也该是显示 cc.openhome. Hello$1@15db9742 与 cc.openhome.Hello$2@6d06d69c 之类的信息，事实上，执行结果会是显示两次"Hello, world!"。也就是说，Lambda 表达式中 this 的参考对象以及 toString()(也就是 this.toString()) 的接受者，是来自 Lambda 的周围环境(Context)，也就是看 Lambda 表达式是在哪个名称范畴(Scope)，就能参考该范畴内的名称，如变量或方法。

在上面的范例中，因为是 Hello 类包围了 Lambda 表达式，Lambda 表达式可以参考类范畴中的名称，范例中定义了 Hello 类的 toString()返回"Hello, world!"字符串，因而执行时才会显示两次"Hello, world!"。

在 7.2.2 节中介绍过，JDK8 出现前，如果要在匿名内部类中存取局部变量，则该局部变量必须是 final，否则会发生编译错误。而在 JDK8 之后，如果变量本身等效于 final 局部变量，也就是说，如果变量不会在匿名类中有重新指定的动作，就可以不用加上 final 关键词。例如以下代码在 JDK8 之后不会有错：

```
String[] names = {"Justin", "Monica", "Irene"}; // JDK8 前必须加上 final
Runnable runnable = new Runnable() {
    public void run() {
        for(String name : names) {
            out.println(name);
        }
    }
};
```

因为 Runnable 符合函数接口定义，因此上例可以改为 Lambda 表达式：

```
String[] names = {"Justin", "Monica", "Irene"};
Runnable runnable = () -> {
    for(String name : names) {
        out.println(name);
    }
};
```


非卖品，仅供非商业用途或交流学习使用

如果 Lambda 表达式中捕获的局部变量本身等效于 **final** 局部变量，可以不用在局部变量上加上 **final**。但是，在 Lambda 表达式中可以改变被捕获的局部变量值吗？答案是不行！

JDK 特意禁止用户在 Lambda 中修改局部变量的值，因为 JDK 想要采用 Lambda 的理由之一，是想进一步支持并行程序设计。Lambda 表达式中可变动的局部变量，通常也表示在并行程序中，可能必须处理同步锁定问题，JDK 以禁止用户在 Lambda 中修改局部变量值来避免这类的问题。

12.1.4 方法与构造函数参考

当你临时想要为函数接口定义操作时，Lambda 表达式确实是很方便。然而有时候，你会发现某些静态方法的本身操作流程，与你自行定义的 Lambda 表达式相同。JDK 考虑到这种状况，Lambda 表达式只是定义函数接口操作的一种方式，除此之外，只要静态方法的方法签署中，参数与返回值定义相同，也可以使用静态方法来定义函数接口操作。

举例来说，在 12.1.1 节中曾定义过以下程序代码：

```
package cc.openhome;

public class StringOrder {
    public static int byLength(String s1, String s2) {
        return s1.length() - s2.length();
    }
    ...
}
```

如果想要定义 `Comparator<String>`的操作，必须操作其定义的 `int compare(String s1, String s2)`方法，你可以使用 Lambda 表达式定义：

```
Comparator<String> byLength  = (s1, s2) -> s1.length() - s2.length();
```

然而仔细观察，除了方法名称之外，`StringOrder` 的静态方法 `byLength` 的参数、返回值，与 `Comparator<String>`的 `int compare(String s1, String s2)`的参数、返回值都相同，用户可以让函数接口的操作参考 `StringOrder` 的静态方法 `byLength`：

```
Comparator<String> byLength = StringOrder::byLength;
```

这样的特性称为方法参考(Method Reference)，可以避免用户到处写下 Lambda 表达式，尽量运用现有的 API 操作，也可以改善可读性，在 12.1.1 节中就介绍过，与其写下如下代码：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, (name1, name2) -> name1.length() - name2.length());
```

不如写下如下代码来得清楚：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, StringOrder::byLength);
```

除了参考静态方法作为函数接口操作之外，还可以参考特定对象的实例方法。例如，在 9.1.7 中介绍过，JDK8 在 `Iterable` 上新增了 `forEach()`方法，可以让用户迭代对象进行指定处理：

```
List<String> names = List.of("Justin", "Monica", "Irene");
names.forEach(name -> out.println(name));
new HashSet(names).forEach(name -> out.println(name));
new ArrayDeque(names).forEach(name -> out.println(name));
```


非卖品，仅供非商业用途或交流学习使用

发现了吗？写了三个重复的 Lambda 表达式，forEach()接受 java.util.function.Consumer 接口的实例，Consumer 接口必须操作 void accept(T t)方法，out 是 PrintStream 实例，println()其实是 out 的实例方法，实际上 println()的方法签署与 accept()方法相同，可以直接参考 out 的 println()方法：

```
List<String> names = List.of("Justin", "Monica", "Irene");
names.forEach(out::println);
new HashSet(names).forEach(out::println);
new ArrayDeque(names).forEach(out::println);
```

函数接口操作也可以参考类上定义的非静态方法，函数接口会试图用第一个参数作为方法接收者，而之后的参数依序作为被参考的非静态方法之参数。举例来说：

```
Comparator<String> naturalOrder = String::compareTo;
```

虽然 Comparator<String>的 int compare(String s1, String s2)方法必须有两个参数，然而在以上的方法参考中，会试图用第一个参数 s1 作为 compareTo()的方法接收者，而之后的参数只剩下 s2，刚好作为 s1.compareTo(s2)，实际的应用在 12.1.1 节中也看过：

```
String[] names = {"Justin", "caterpillar", "Bush"};
Arrays.sort(names, String::compareTo);
...
Arrays.sort(names, String::compareToIgnoreCase);
```

方法参考用来重用现有 API 的方法流程，JDK 还提供了构造函数参考(Constructor Reference)，用来重用现有 API 的对象建构流程。你也许会发出疑问：“构造函数？它们有返回值类型吗？”语法上不需要，但事实上有。其实每个构造函数都会有返回值类型，也就是定义它们的类本身。例如，定义了一个 map()方法如下：

```
static <T, R> List<R> map(List<T> list, Function<T, R> mapper) {
    List<R> mapped = new ArrayList<>();
    for(int i = 0; i < list.size(); i++) {
        mapped.add(mapper.apply(list.get(i)));
    }
    return mapped;
}
```

其中使用了 java.util.function.Function 接口，这个接口定义了一个 R apply(T t)方法必须操作，map()方法可以 Function 实例，指定如何将 T 转换为 R，例如，将用户名称转为 Person 实例：

```
List<String> names = List.of(args);
List<Person> persons = map(names, name -> new Person(name));
```

实际上，这不过是将 name 用来调用 Person 的构造函数，那么不如直接参考 Person 的构造函数：

Lambda MethodReferenceDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.*;
import java.util.function.Function;

class Person {
```

```
        String name;

        Person(String name) {
            this.name = name;
        }

        public String toString() {
            return "Person{" + "name=" + name + '}';
        }
    }

    public class MethodReferenceDemo {
        static <P, R> List<R> map(List<P> list, Function<P, R> mapper) {
            List<R> mapped = new ArrayList<>();
            for(int i = 0; i < list.size(); i++) {
                mapped.add(mapper.apply(list.get(i)));
            }
            return mapped;
        }

        public static void main(String[] args) {
            List<String> names = List.of(args);
            List<Person> persons = map(names, Person::new);
            persons.forEach(out::println);
        }
}
```

如果某类有多个构造函数，就会使用函数接口的方法来比较，找出对应的构造函数进行调用。这个范例也示范了刚才介绍的方法参考，`forEach()`接受 `Consumer` 实例，而 `Consumer` 的操作直接参考了 `System.out` 的 `println()`方法。

12.1.5 接口默认方法

到目前为止可以看到， Lambda 不仅只是引入了 Lambda 语法，也考虑了如何使用现有的 API。除此之外，JDK 也试图在现有 API 增加功能，让开发者在迁移至 JDK8 之后的平台的同时，马上就有更多能搭配 Lambda 的强大 API 可使用。

这里的问题是，这些 API 要放在哪？例如，迭代对象时的 `forEach()`方法要放在哪呢？把这些方法定义为 `Collections` 类上的 `static` 方法。过去不少为增强 `Collection` 功能的第三方程式库就采取类似作法，然而，JDK 希望这些 API 具备面向对象程序设计风格，在撰写程序代码时也能更为流畅，因而将这些方法定义为 `Collections` 类的 `static` 方法并不适合，也就是说，JDK 希望的风格会是：

```
List<String> names = ...;
names.filter(s -> s.length() < 3)
     .forEach(out::println);
```

而不像是以下的风格：

```
forEach(filter(names, s -> s.length() < 3), out::println);
```

前者的风格显然在阅读上较为流畅。只是，没有办法在 `List` 接口中增加像 `filter` 之类的方法。如果用的是 JDK7 或之前的版本，答案当然是否定的！所有操作 `List` 接口的客户端程序代码都会出错，因为它们本来就没有操作新增的那些方法。建立一个新的 Collections2 API 是个

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

选项，不过现有的 Collection API 遍布在全世界许多的链接库中，要把这些既有的 Collection API 替换为新的 Collections2 是个庞大任务，在 JDK8 发布后，开发者应该不会想马上用新的 Collections2 API 吧！

最后采取的策略是，直接演变 interface 的语法，在 JDK8 中，interface 定义时可以加入默认操作，或者称为默认方法(Default Method)。默认方法的实例之一，就是定义在 Iterable 接口的 forEach() 方法：

```
package java.lang;

import java.util.Iterator;
import java.util.Objects;
import java.util.function.Consumer;

public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

默认方法必须使用 default 关键字修饰，默认权限为 public，而由于 iterator() 方法并没有操作 Iterable 的操作类，必须操作 iterator() 方法，这么一来，API 客户端就可以直接使用 forEach() 方法。例如，程序代码：

```
List<String> names = List.of("Justin", "caterpillar", "Monica");
names.forEach(out::println);
```

因为 forEach() 方法本身已有操作，所以不会破坏 Iterable 现有的其他操作。默认方法令接口看来像是有抽象方法的抽象类，不过不同点在于，默认方法中不能使用数据成员，因为接口本身不能定义数据成员，也就是默认方法中不能有直接变更状态的流程。

默认方法也给了共享相同操作的方便性。例如，定义自己的 Comparable 接口：

```
public interface Comparable<T> {
    int compareTo(T that);

    default boolean lessThan(T that) {
        return compareTo(that) < 0;
    }
    default boolean lessOrEquals(T that) {
        return compareTo(that) <= 0;
    }
    default boolean greaterThan(T that) {
        return compareTo(that) > 0;
    }
    ...
}
```

如果有个 Ball 类打算操作这个自定义的 Comparable 接口，就只需要操作 compareTo() 方法：

```
public class Ball implements Comparable<Ball> {
    private int radius;
    ...
    public int compareTo(Ball that) {
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
            return this.radius - that.radius;
        }
    }
```

这么一来，每个 Ball 实例都可拥有 Comparable 接口定义的那些默认方法。因为类可以操作多个接口，默认方法的新特性让你可以在某些接口中定义可共享的一些操作。如果有个类需要某些可共享的操作，就只需要操作相关接口，并操作接口中未操作的抽象方法，那么就可以混入(Mix-in)这些共享的操作了。

在操作默认方法时，可能会将算法定义为数个更小的流程，而这些小流程并不用公开给外界。基于此需求，从 JDK9 开始，接口中也可以定义 private 方法了，虽然会被公开的默认方法调用，不过不用加上 default 修饰。例如：

```
public interface Some {
    default void doIt() {
        subMethod1();
        subMethod2();
    }
    private void subMethod1() {
        // 私有操作...
    }
    private void subMethod2() {
        // 私有操作...
    }

}
```

1. 辨别方法的实作版本

JDK8 出现前不让接口拥有默认方法是有原因的，因为操作接口是广义的多重继承，接口没有操作时，类与接口继承时的方法来源在判断上就会单纯许多。接口在 JDK8 中允许有默认操作，引入了强大的威力，也引入了更多的复杂度，读者得留意到底采用的是哪个方法版本。

就如同子类可继承父类操作，接口也可以被继承，而抽象方法或默认方法都会被继承下来，在子接口中再以抽象方法重新定义一次父接口已定义的抽象方法，通常是为了文件化，这是过去经常看到的实践(Practice)，反正没有操作，没什么必须辨别操作版本的问题。

一旦接口中可以定义默认方法，辨别操作版本时就有许多需要注意的地方。例如，父接口中的抽象方法可以在子接口中以默认方法操作，父接口中的默认方法可以在子接口中被新的默认方法重新定义。

如果父接口中有个默认方法，子接口中再度声明与父接口默认方法相同的方法命名，但没有写出 default，也就是没有方法操作，那么子接口中该方法就直接重新定义了父类中的默认方法操作为抽象方法。例如，自定义一个 BiIterable：

```
import java.util.Iterator;
import java.util.function.Consumer;

public interface BiIterable<T> extends Iterable<T> {
    Iterator<T> iterator();
    void forEach(Consumer<? super T> action);
    ...
}
```

在上面的例子中，BiIterable 的 forEach()方法就没有操作了，操作 BiIterable 的类就必

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

须操作 forEach()方法。如果有两个父接口都定义了相同方法命名的默认方法，那么会引发冲突。例如，假设 Part 与 Canvas 两个接口都有个 default 的 draw()方法，而 Lego 接口在继承 Part、Canvas 时，没有重新定义 draw()，就会发生编译错误。

解决的方法是明确重新定义 draw()，如重新定义为抽象或默认方法。在重新定义为默认方法时，想明确调用某个父接口的 draw()方法，必须使用接口名称与 super 明确指定，例如：

```
public interface Lego extends Part, Canvas {
    default void draw() {
        Part.super.draw();
    }
}
```

类在操作有默认方法的接口时，可以重新定义默认方法，无论是重新定义具体操作，或是将该方法标示为 abstract。如果操作时有两个接口都定义了相同方法签署的默认方法，那么会引发冲突，解决的方法是明确重新定义 draw()，如重新定义为抽象或默认方法。在重新定义为具体方法时，想明确调用某个接口的 draw()方法，也是得使用接口名称与 super 明确指定。

如果类操作的两个接口拥有相同的父接口，其中一个接口重新定义了父接口的默认方法，而另一个接口没有，那么操作类会采用重新定义了的版本。例如，若自定义一个 LinkedList 如下：

```
class LinkedList<E> implements List<E>, Queue<E>
```

若自定义的 List 与 Queue 父接口 Collection 定义了 removeAll()方法，List 继承自 Collection 重新以默认方法定义了 removeAll()，Queue 继承自 Collection，没有重新定义 removeAll()方法，那么 LinkedList 采用的版本就是 List 中的 removeAll()默认方法，而不是 Collection 中的 removeAll()默认方法。

如果子类继承了父类同时操作某接口，而父类中的方法与接口中的默认方法具有相同方法命名，则采用父类的方法定义。

简单来说，类中的定义优先于接口中的定义，如果有重新定义，就以重新定义的为主，必要时使用接口与 **super** 指定采用哪个默认方法。

JDK8 除了让接口可以定义默认方法之外，也开始允许在接口中定义静态方法，实际上在 9.1.8 节中使用到的 nullsFirst()、reverseOrder()等方法，就是定义在 Comparator 接口上的静态方法。

同样地，接口中公开的静态方法，其演算流程可能会被拆解为数个小流程，定义与其他静态方法之中，若这些方法不用公开给外界，那么从 JDK9 开始，可以定义为 private 的静态方法了。

2. 回顾 Iterable、Iterator 和 Comparator

来看看 Java SE 标准 API 中的几个具有默认方法的接口，其中一个在 9.1.7 节中看过，也就是 Iterable 接口，它新增了 forEach()默认方法：

```
...
public interface Iterable<T> {
    ...
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
    ...
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

之前在一些范例中看过 forEach()的应用了，也就是将收集的对象逐一迭代，不用再通过 Iterator 实例来进行外部迭代。实际上，Iterator 也有个 forEachRemaining()的默认操作，可以用来迭代剩余元素：

```
...
public interface Iterator<E> {
    ...
    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

在 9.1.8 节曾经介绍过 Comparator，当时使用过它上头定义的 nullsFirst()、reverseOrder()等静态方法，实际上，Comparator 也定义了一些默认方法，例如 thenComparing()方法，这可以让用户用更具弹性的方式，从现有的 Comparator 实例组合出更复杂的 Comparator 实例。例如，你可能想要排序时先按客户的姓来排，如果姓相同再按名来排，如果姓名都相同，再按他们居住地的邮政编码来排，那么你可以如下建立 Comparator：

Lambda CustomerDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.*;
import static java.util.Comparator.comparing;

public class CustomerDemo {
    public static void main(String[] args) {
        List<Customer> customers = List.of(
                new Customer("Justin", "Lin", 804),
                new Customer("Monica", "Huang", 804),
                new Customer("Irene", "Lin", 804)
        );

        Comparator<Customer> byLastName = comparing(Customer::getLastName);

        customers.sort(
                byLastName
                .thenComparing(Customer::getFirstName)
                .thenComparing(Customer::getZipCode)
        );

        customers.forEach(out::println);
    }
}

class Customer {
    private String firstName;
    private String lastName;
    private Integer zipCode;

    public Customer(String firstName, String lastName, Integer zipCode) {
        this.firstName = firstName;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        this.lastName = lastName;
        this.zipCode = zipCode;
    }

    public String toString() {
        return String.format("Customer(%s %s, %d)", firstName, lastName, zipCode);
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public Integer getZipCode() {
        return zipCode;
    }
}
```

每次 Comparator 实例调用 thenComparing()方法，都会返回新的 Comparator 实例，你就可以再调用 thenComparing()方法，组合出自己想要的排序方式。程序的执行结果如下：

```
[Customer(Monica Huang, 804), Customer(Irene Lin, 804), Customer(Justin Lin, 804)]
```

12.2 Functional 与 Stream API

在之前的介绍中， Lambda 不仅只有语法上的新功能，也有与现有 API 的兼容性，并增强现有 API 功能。除此之外，Lambda 项目还包括了全新的 API，这些 API 主要位于 **java.util.function** 与 **java.util.stream** 套件之中，了解并使用这些新 API，才能真正发挥 Lambda 的威力。

12.2.1 使用 Optional 取代 null

JDK8 新增了 java.util.Optional 类，在介绍这个类如何使用之前，必须先引用一下 Java Collection API 及 JSR166 参与者之一 Doug Lea 的话：

Null sucks.

图灵奖得主、快速排序发明者 Tony Hoare，在 QCon London 2009 主讲 *Null References: The Billion Dollar Mistake* 场次时也谈到 null：

I call it my billion-dollar mistake.

在使用 Java 开发一段时间之后你就会(或已经)发现，你经常会与 **NullPointerException** 奋战，如果有个变量参考至 null，就会引发这个异常。

null 的最根本问题在于语意含糊不清，虽然就字面来说，null 可以是“不存在”“没有”“无”或“空”的意思，因此在应用时，总是令人感到模棱两可，也就让开发者有了各自解释的空间，当开发者想到“嘿！这边可以没有东西……”就直接放个 null，或者是想到“嗯！没什么东西可以返回……”，就不假思索地返回个 null，然而用户却总是忘了检查 null，引发各种可能的错误。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

由于 null 的根本问题在于含糊而不明确，要避免使用 null 的方式，就是确认使用 null 的时机与目的，并使用明确的语义。在过去使用 null 的情况中，开发者在方法中返回 null，通常代表着客户端必须检查是否为 null，并在 null 的情况下使用默认值，以便后续程序继续执行。举个例子来说：

```
public static void main(String[] args) {
    String nickName = getNickName("Duke");
    if (nickName == null) {
        nickName = "Openhome Reader";
    }
    out.println(nickName);
}

static String getNickName(String name) {
    // 假装的键值数据库
    Map<String, String> nickNames = Map.of(
        "Justin", "caterpillar",
        "Monica", "momor",
        "Irene", "hamimi"
    );
    return nickNames.get(name); // 键不存在时会返回 null
}
```

在上面的程序中，如果调用 getNickName()时忘了检查返回值是否为 null，那么执行结果就会直接显示 null，在这个简单的例子中并不会怎样，只是显示结果令人困惑罢了，但如果后续的执行流程牵涉至关重要的结果，程序快乐地继续执行下去，错误可能到最后才会呈现出来。

可以将 getNickName()修改使其一定会返回 Optional<String>实例，而不是返回 null，调用方法时如果返回类型是 **Optional**，应该立即想到它可能包含也可能不包含值。要建立 Optional 实例有几个静态方式，使用 **of()**方法可以指定非 null 值建立 Optional 实例，使用 **empty()**方法可以建立不包含值的 Optional 实例。例如，可使用 Optional 来改写前面的 getNickName()方法：

```
static Optional<String> getNickName(String name) {
    Map<String, String> nickNames = Map.of(
        "Justin", "caterpillar",
        "Monica", "momor",
        "Irene", "hamimi"
    );
    String nickName = nickNames.get(name);
    return nickName == null ? Optional.empty() : Optional.of(nickName);
}
```

因为调用 getNickName()时返回的是 Optional 类型实例，语义上表示它包含也可能不包含值，客户端就要意识到必须对是否包含值进行检查，如果不检查就直接调用 Optional 的 get()方法：

```
String nickName = getNickName("Duke").get();
out.println(nickName);
```

在 Optional 没有包含值的情况下，就会直接抛出 **java.util.NoSuchElementException**，这实现了速错(Fail Fast)的概念，可以让开发者可以立即发现错误，并了解到必须使用程序代码做些检查，可能的方式之一就如：

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
Optional<String> nickOptional = getNickName("Duke");
String nickName = "Openhome Reader";
if(nickOptional.isPresent()) {
    nickName = nickOptional.get();
}
out.println(nickName);
```

不过这看来有点啰唆，可以使用 orElse() 方法，指定值不存在时的替代值：

```
Optional<String> nickOptional = getNickName("Duke");
out.println(nickOptional.orElse("Openhome Reader"));
```

过去许多链接库中使用了不少 null，这些链接库无法说改就改，可使用 Optional 的 **ofNullable()** 来衔接链接库中会返回 null 的方法。使用 ofNullable() 方法时，若指定了非 null 值就会调用 of() 方法，指定了 null 值就会调用 empty() 方法。例如，前面的 getNickName() 方法可以更简洁地修改为：

```
static Optional<String> getNickName(String name) {
    Map<String, String> nickNames = Map.of(
        "Justin", "caterpillar",
        "Monica", "momor",
        "Irene", "hamimi"
    );
    return Optional.ofNullable(nickNames.get(name));

}
```

Optional 还有更高级的 map() 与 flatMap() 方法，这稍后就会介绍。在这之前，得先认识一下 java.util.function 套件中各个函数接口。

12.2.2 标准 API 的函数接口

Lambda 表达式实际的类型要看函数接口而定，虽然可以自行定义函数接口，只不过对于几种常用的函数接口行为。JDK8 已经定义了几个通用的函数接口，你可以先基于这些通用函数接口来撰写程序，在必要时再考虑自定义函数接口。通用函数接口放置于 **java.util.function** 套件之中。就行为来说，可以分为 Consumer、Function、Predicate 与 Supplier 四个类型。

1. Consumer 函数接口

如果需要的行为是接受一个自变量，然后处理后不返回值，就可以使用 Consumer 接口，它的定义是：

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    ...
}
```

接受 Consumer 方法的实际例子就是 Iterable 上的 forEach() 方法：

```
default void forEach(Consumer<? super T> action) {
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
```

既然接受了自变量但没有返回值，这行为就像纯粹消耗了自变量，也就是命名为 Consumer 的原因。如果真的有结果产生，就是以副作用(Side Effect)形式呈现，就像改变某对象状态，或者是进行了输入/输出。例如，使用 System.out 的 println()进行输出：

```
List.of("Justin", "Monica", "Irene").forEach(out::println);
```

Consumer 接口主要是接受单一对象实例作为自变量，对于基本类型 int、long、double，另外有 IntConsumer、LongConsumer、DoubleConsumer 三个函数接口；对于接受两个对象实例作为自变量的接口则为 BiConsumer，另外还有 ObjIntConsumer、ObjLongConsumer、ObjDoubleConsumer，这三个函数接口第一个参数接受对象实例，第二个参数分别接受 int、long 与 double。

2. Function 函数接口

如果需要的是接受一个自变量，然后以该自变量进行计算后返回结果，就可以使用 Function 接口，它的定义是：

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    ...
}
```

因为这行为就像是数学函数 $y=f(x)$，给予 x 值计算出 y 值的概念，因此命名为 Function。应用的例子之一曾经在 12.1.4 节的 MethodReferenceDemo 范例中介绍过，该范例的 map()方法就接受 Function 实例，可以指定如何将收到的值转换为另一个值。

Function 的子接口为 UnaryOperator，特殊化为参数与返回值都是相同类型，虽然 JDK8 之后仍不支持运算符重载，不过这个命名显然源自于某些语言中，运算符也是个函数的概念：

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,T>
```

对于基本类型的函数转换，则有着 IntFunction、LongFunction、DoubleFunction、IntToDoubleFunction、IntToLongFunction、LongToDoubleFunction、LongToIntFunction、DoubleToInt- Function、DoubleToLongFunction 等函数接口，看看它们的名称或 API 文件，作用应该都一目了然。

如果需要接受两个自变量然后返回一个结果，则可以使用 BiFunction：

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
    ...
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

类似地，BinaryOperator 是 BiFunction 的子接口，特殊化为两个参数与返回值都是相同类型。对于基本类型，也有一些对应的函数接口，只要是 BiFunction 或是 BinaryOperator 名称结尾的，都是类似的东西，读者可以直接查询 API 来了解。

3. Predicate 函数接口

如果接受一个自变量，然后只返回 boolean 值，也就是根据传入的自变量直接论断真假的行为，就可以使用 Predicate 函数接口，其定义为：

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    ...
}
```

举例来说，如果有个文件名的 String 数组 fileNames，想要知道其中扩展名为.txt 的有几个，可以如下撰写：

```
long count = Stream.of(fileNames)
                   .filter(name -> name.endsWith("txt"))
                   .count();
```

后面还会详细介绍 Stream，此实例的 filter()方法接受 Predicate 实例，每个元素都会由 Predicate 来判断是否被过滤出来保留。类似地，BiPredicate 是接受两个自变量，返回 boolean 值，基本类型对应的函数接口则有 IntPredicate、LongPredicate、DoublePredicate。

4. Supplier 函数接口

如果需要的行为是不接受任何自变量，然后返回值，那可以使用 Supplier 函数接口：

```
package java.util.function;

@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

既然不接受自变量就能返回值，返回值的来源就有几个可能性，如提供容器、固定值、某个时间某个事物的状态值、某个外部输入值、某个要按需(On-demand)索取的(昂贵)运算等。举个例子，稍后就会介绍的 Stream 接口，定义有 collect()方法，其中有个版本就接受 Suppiler 实例，其作用是提供容器作为收集对象之用，如图 12.1 所示。

至于那些 BooleanSupplier、DoubleSupplier、IntSupplier、LongSupplier，读者可以直接查询一下 API，就能了解其作用。

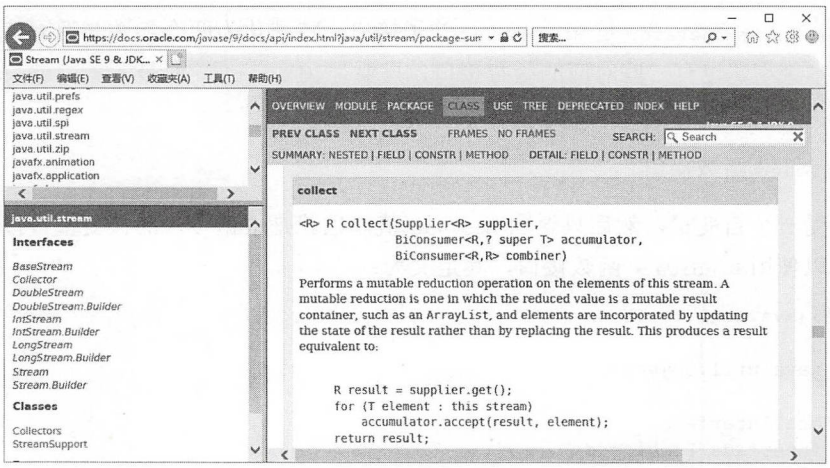


图 12.1 Stream 接口的 collect()方法

12.2.3 使用 Stream 进行管道操作

在正式了解 Stream 接口的作用之前，先来看一个程序片段：

```
String fileName = args[0];
String prefix = args[1];
String firstMatchdLine = "no matched line";
for (String line : Files.readAllLines(Paths.get(fileName))) {
    if(line.startsWith(prefix)) {
        firstMatchdLine = line;
        break;
    }
}
out.println(firstMatchdLine);
```

程序中使用到 java.nio.file 的 Files 与 Paths 类，它们是第 14 章将介绍到的 NIO2 中的标准类，get()方法返回 Path 实例，代表指定的路径，readAllLines()方法读取档案全部内容，并以换行为依据，将每行内容收集在 List<String>后返回，程序会找到第一个符合条件的行，然后显示小写后离开循环。在 JDK8 之后，这类的需求，建议改用以下的程序来完成：

Lambda LineStartsWith.java

```
package cc.openhome;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Optional;

public class LineStartsWith {
    public static void main(String[] args) throws IOException {
        String fileName = args[0];
        String prefix = args[1];
        Optional<String> firstMatchdLine =
                Files.lines(Paths.get(fileName))
                     .filter(line -> line.startsWith(prefix))
                     .findFirst();
```

```
            System.out.println(firstMatchdLine.orElse("no matched line"));
        }
}
```

可以看出最大的差别是没有用到 for 循环与 if 判断式，以及使用了管道(Pipeline)操作风格，而功能上也有所差异，如果读取的文件很大，第二个程序片段会比第一个程序片段更加有效。

Files 的 lines()方法，会返回 **java.util.stream.Stream** 实例，就这个例子来说是 Stream<String>，使用 Stream 的 filter()方法会过滤留下符合条件的元素，findFirst()方法会尝试看看留下的元素有没有首元素，因为也可能完全没有元素，因此返回 Optional<String>实例。

功能上的差异性在于，第一个程序片段的 Files.readAllLines()方法返回的是 List<String>实例，其中包括了文档中所有行，如果第一行就符合指定的条件了，那后续的行读取就是多余的；第二个程序片段的 lines()方法实际上没有进行任何一行的读取，filter()也没有做任何一行的过滤，直到调用 findFirst()时，filter()指定的条件才会真正去执行，而此时才会要求 lines()返回的 Stream 进行第一行读取，如果第一行就符合，那后续的行就不会再读取，效率的差异性就在于此。

之所以能够达到这类惰性求值(Lazy Evaluation)的效果，也就是需要时 findFirst()要求 filter()，而 filter()再要求读取文档下一行，这种“你需要我再给”的行为就是 Stream 实例。第一个程序片段要取得 List 返回的 Iterator，以搭配 for 循环进行外部迭代(External Iteration)，第二个程序片段则将迭代行为隐藏在 lines()、filter()与 findFirst()方法之中，称为内部迭代(Internal Iteration)，因为内部迭代的行为是被隐藏的，因此增加了很多可以提高效率的可能性。

Stream 的顶层父接口是 AutoClosable，而直接父接口 java.util.stream.BaseStream 的 close()操作了 close()方法。然而基本上绝大多数的 **Stream** 并不需要调用 **close()**方法，除了一些 I/O 操作之外，例如 **Files.lines()**、**Files.list()**与 **Files.walk()**方法，建议这类操作可以搭配尝试关闭资源(try-with-resource)语法。

JDK8 引入了 Stream API，也引入了管道操作风格，一个管道基本上包括了几个部分：

- 来源(Source)。
- 零或多个中间操作(Intermediate Operation)。
- 一个最终操作(Terminal Operation)。

来源可能是文档、数组、群集、产生器(Generator)等，在这个例子中就是指定的文档。中间操作又称为集合操作(Aggregate Operation)，这些操作调用时，并不会立即进行手边的数据处理，它们很懒惰(Lazy)，只会在后续中间操作要求数据时才会动手处理下一笔数据，例如第二个程序片段中的 filter()方法，最终操作是最后真正需要结果的操作，这个操作会要求之前懒惰的中间操作开始动手。

这就是 Stream API 之所以命名为 Stream 的原因，Stream 实例衔接了来源，每个中间操作方法都会返回 Stream 实例，但不会实际进行数据处理，每个中间操作后的 Stream 实例会串联在一起，Stream 提供的最终操作方法不是返回 Stream，而是返回真正需要的结果，最终操作方法会引发先前中间操作时，串连在一起的 Stream 实例进行数据处理。

实际上从来源进行一些运算，以求得最终结果，正是程序设计时最常进行的动作，因此 JDK8 在不少具有来源概念的 API 上，都增加了可返回 Stream 的方法，除了这边看到的 Files 之外，你还可以使用 Stream 上的静态方法来建立 Stream 实例，例如 of()方法，对于数组也可以使用 Arrays 的 stream()方法来建立 Stream 实例。

Collection 也是个例子，其定义了 stream()方法会返回 Stream 实例，只要是 Collection 都可以进行中间操作。例如，原本有个程序片段：

```
List<Player> players = ...;
List<String> names = new ArrayList<>();
for(Player player : players) {
    if(player.getAge() > 15) {
        names.add(player.getName().toUpperCase());
    }
}
for(String name : names) {
    System.out.println(name);
}
```

在JDK8之后可以改为以下的风格：

Lambda PlayerDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.Arrays;
import java.util.List;
import static java.util.stream.Collectors.toList;

public class PlayerDemo {
    public static void main(String[] args) {
        List<Player> players =List.of(
                new Player("Justin", 39),
                new Player("Monica", 36),
                new Player("Irene", 6)
        );
        players.stream()
                .filter(player -> player.getAge() > 15)
                .map(Player::getName)
                .map(String::toUpperCase)
                .collect(toList())
                .forEach(out::println);
    }
}

class Player {
    private String name;
    private Integer age;

    public Player(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }
}
```

每个中间操作都隐藏了细节，除了增加更多效率改进的空间之外，也鼓励开发者多利用这类风格，来避免撰写一些重复流程，或思考目前的复杂演算中，实际上会是由哪些小任务完成。

例如，在 for 循环中使用了 if：

```
for(Player : players) {
    if(player.getAge() > 15) {
        // 这是下一个小任务
    }
}
```

也许就有改用 filter() 方法的可能性：

```
players.stream()
.filter(player -> player.getAge() > 15)
... // 接下来的中间操作或最终操作
```

如果在 for 循环中从一个类型对应至另一个类型：

```
for(Player player: players) {
    String upperCase = player.getName().toUpperCase();
    ...下一个小任务
}
```

也许就有改用 map() 方法的可能性：

```
players.stream()
       .map(Player::getName)
       .map(String::toUpperCase)
       ...下一个小任务
```

许多时候，for 循环中就渗杂了许多小任务，从而使 for 循环中的程序代码艰涩难懂。辨识出这些小任务，运用中间操作，形成管道化操作风格，就能增加程序代码阅读时的流畅性。

Stream 的直接父接口为 BaseStream，而 BaseStream 还有 DoubleStream、IntStream 与 LongStream 这三个用于基本类型操作的子接口。

Stream 只能迭代一次，重复对 Stream 进行迭代，会引发 IllegalStateException。

12.2.4 进行 Stream 的 reduce 与 collect

从一组数据按条件求得一个数，或将一组数据按条件收集至另一个容器，程序设计中不少地方都存在这类需求，使用循环解决这类需求，是许多开发者最常采用的方法。举例来说，求得一组员工的男性平均年龄：

```
List<Employee> employees = ...;
int sum = 0;
for(Employee employee : employees) {
    if(employee.getGender() == Gender.MALE) {
        sum += employee.getAge();
    }
}
int average = sum / employees.size();
```

循环中有进行过滤的动作，而若要求得一组员工的男性最大年龄，可能是这样撰写：

```
int max = 0;
for(Employee employee : employees) {
    if(employee.getGender() == Gender.MALE) {
```

```
            if(employee.getAge() > max) {
                max = employee.getAge();
            }
        }
    }
```

实际上，循环中也有进行过滤的动作，你的程序中这类需求都存在着类似的流程结构，而你也不断重复撰写着类似结构，而且从阅读程序代码角度来看，无法一眼察觉程序意图，在 JDK8 之后，可以改写为：

Lambda EmployeeDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.Arrays;
import java.util.List;

public class EmployeeDemo {
    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Justin", 39, Gender.MALE),
                new Employee("Monica", 36, Gender.FEMALE),
                new Employee("Irene", 6, Gender.FEMALE)
        );

        int sum = employees.stream()
                .filter(employee -> employee.getGender() == Gender.MALE)
                .mapToInt(Employee::getAge)
                .sum();

        int average = (int) employees.stream()
                .filter(employee -> employee.getGender() == Gender.MALE)
                .mapToInt(Employee::getAge)
                .average()
                .getAsDouble();

        int max = employees.stream()
                .filter(employee -> employee.getGender() == Gender.MALE)
                .mapToInt(Employee::getAge)
                .max()
                .getAsInt();

        List.of(sum, average, max).forEach(out::println);
    }
}

enum Gender { FEMALE, MALE }

class Employee {
    private String name;
    private Integer age;
    private Gender gender;

    public Employee(String name, Integer age, Gender gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
```

```
    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }

    public Gender getGender() {
        return gender;
    }
}
```

除此之外，IntStream 也提供了 sum()、average()、max()、min()等方法，那么如果有其他的计算需求呢？

1. 使用 reduce()方法

观察前面的循环结构，实际上有个步骤都是将一组数据逐步取出削减，然而通过指定运算以取得结果的结构，JDK8 将这个流程结构通用化，定义了 **reduce()**方法来达到自定义运算需求。例如，以上三个流程，也可以使用 reduce()重新撰写如下：

Lambda EmployeeDemo2.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.Arrays;
import java.util.List;

public class EmployeeDemo2 {
    public static void main(String[] args) {
        List<Employee2> employees = List.of(
                new Employee2("Justin", 39, Gender2.MALE),
                new Employee2("Monica", 36, Gender2.FEMALE),
                new Employee2("Irene", 6, Gender2.FEMALE)
        );

        int sum = employees.stream()
                .filter(employee -> employee.getGender() == Gender2.MALE)
                .mapToInt(Employee2::getAge)
                .reduce((total, age) ->  total + age)
                .getAsInt();

        long males = employees.stream()
                .filter(employee -> employee.getGender() == Gender2.MALE)
                .count();

        long average = employees.stream()
                .filter(employee -> employee.getGender() == Gender2.MALE)
                .mapToInt(Employee2::getAge)
                .reduce((total, age) ->  total + age)
                .getAsInt() / males;

        int max = employees.stream()
                .filter(employee -> employee.getGender() == Gender2.MALE)
                .mapToInt(Employee2::getAge)
```

```
                .reduce(0, (currMax, age) -> age > currMax ? age : currMax);

        List.of(sum, average, max).forEach(out::println);
    }
}

enum Gender2 { FEMALE, MALE }

class Employee2 {
    ...
}
```

给 reduce()的 Lambda 表达式必须接受两个自变量，第一个自变量为访问该组数据上一元素后的运算结果，第二个自变量为目前访问元素，Lambda 表达式本身就是你原先在循环中打算进行的运算；reduce()若没有指定初值(严格说是恒等式，见 API 文件)，就会试着使用该组数据中第一个元素作为第一次调用 Lambda 表达式时的第一个自变量值，因为考虑到数据组可能为空，因此 reduce()不指定初值的版本，会返回 OptionalInt(非基本类型数据组，则会是 Optional)。

2. 使用 collect()方法

那么，如何将一组员工的男性收集至另一个 List<Employee>呢？在 employees.stream().filter(employee -> employee.getGender() == Gender.MALE)之后，返回的是 Stream<Employee>，因为 filter()是 Stream 的中间操作，不是最终操作，使用 reduce()的话，在处理完新元素后，每次都会返回新的计算结果，作为下一次 Lambda 表达式接受的第一个自变量，显然不适合用来收集对象。

使用 Stream 的 collect()方法，以将一组员工的男性收集至另一个 List<Employee>的需求来说，最简单的方式就是：

```
List<Employee> males = employees.stream()
          .filter(employee -> employee.getGender() == Gender.MALE)
          .collect(toList()); // toList()是java.util.stream.Collectors的静态方法
```

在 12.2.3 节中的 PlayDemo 范例也看过 toList()的使用，Collectors 的 toList()方法返回的并不是 List，而是 java.util.stream.Collector 实例。Collector 主要的四个方法是：**suppiler()**返回 Suppiler，定义收集结果的新容器如何建立；**accumulator()**返回 BiConsumer，定义如何使用结果容器收集对象；**combiner()**返回 BinaryOperator，定义若有两个结果容器时如何合并为一个结果容器；**finisher()**返回 Function，选择性地定义如何将结果转换为最后的结果容器。

来看看 Stream 的 collect()方法另一个版本，有助于了解 Collector 这几个方法如何使用。以下的程序片段与上面的 collect()范例结果是相同的：

```
List<Employee> males = persons.stream()
                    .filter(employee -> employee.getGender() == Gender.MALE)
                    .collect(
                         () -> new ArrayList<>(),
                         (maleLt, employee) -> maleLt.add(employee),
                         (maleLt1, maleLt2) -> maleLt1.addAll(maleLt2)
                );
```

当 collect()需要收集对象时，会使用第一个 Lambda 来取得容器对象，这相当于 Collector 的 suppiler()的作用，第二个 Lambda 定义了如何收集对象，也就是 Collector 的 accumulator()

的作用。在使用具有并行处理能力的 Stream 时，有可能会使用多个容器对原数据组进行分而治之(Divide and Conquer)。当每个小任务完成时，该如何合并，就是第三个 Lambda 要定义的。上面代码可以如下来写比较简洁：

```
List<Employee> males = employees.stream()
                .filter(employee -> employee.getGender() == Gender.MALE)
                .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

当然，使用这个版本的 collect()需要处理比较多的细节，可以先看看 Collectors 提供了哪些 Collector 操作，例如收集为 List 的需求，可以如前面使用 toList()取得现成的 Collector 操作对象，或者若想要按性别分组，那可以使用 Collectors 的 groupingBy()方法，告诉它要用哪个当作分组的键(Key)，最后返回的 Map 结果会以 List 作为值(Value)：

```
Map<Gender, List<Employee>> males = employees.stream()
                .collect(groupingBy(Employee::getGender));
```

有的方法也兼具另一种流畅风格，例如，想在按性别分组之后，取得分组下的姓名，那可以如下撰写：

```
Map<Gender, List<String>> males = employees.stream()
                .collect(
                    groupingBy(Employee::getGender,
                        mapping(Employee::getName, toList()))
                );
```

例如，想在按性别分组之后，分别取得男女年龄之和，那可以如下撰写：

```
Map<Gender, Integer> males = employees.stream()
                .collect(
                        groupingBy(Employee::getGender,
                            reducing(0, Employee::getAge, Integer::sum))
                );
```

要求得各性别下平均年龄的话，Collectors 也有个 averagingInt()方法可以使用：

```
Map<Gender, Double> males = employees.stream()
                .collect(
                        groupingBy(Employee::getGender,
                            averagingInt(Employee::getAge))
                );
```

Collectors 上有个 joining()静态方法，如果在管线化操作之后，想要进行字符串的链接，可以使用类似以下方式：

```
List<Customer> customers = ...;
String joinedFirstNames = customers.stream()
                        .map(Customer::getFirstName)
                        .collect(joining(", "));
```

顺便提一下，若只是想单纯地指定每个字符串间以逗号区隔进行连结，JDK8 中的 String 增加了 **join()**静态方法可以直接使用。例如：

```
String message = String.join("-", "Java", "is", "cool"); // 产生 "Java-is-cool"
```

join()接受 CharSequence 操作对象，String 是其中之一。类似地，如果有一组操作了 CharSequence 的对象，想将它们返回的字符串描述以指定字符串链接在一起，有另一个版本的

join()可以达到这项需求。例如：

```
List<String> strs = List.of("Java", "is", "Cool");
String message = String.join("-", strs); // 产生 "Java-is-cool"
```

不仅List，这个版本的join()实际上可接受Iterable操作对象，因此Set等都可以使用这个join()方法。前面的范例实际上只收集字符串，对于这类需求，可以直接使用JDK8增加的StringJoiner类。例如：

```
StringJoiner joiner = new StringJoiner("-");
String message = joiner.add("Java")
                       .add("is")
                       .add("cool")
                       .toString();  // 产生 "Java-is-cool"
```

在JDK9中，Collectors上新增了filtering()方法，可指定过滤条件并返回Collector实例，filtering()方法用来减少管道操作的层次，或者建立可重用的Collector，例如原先有个操作：

```
List<Employee> males = employees.stream()
      .filter(employee -> employee.getGender() == Gender.MALE)
      .collect(toList());
```

若改用filtering()方法，可以如下：

```
List<Employee> males = employees.stream()
     .collect(filtering(
       employee -> employee.getGender() == Gender.MALE, toList())
      );
```

12.2.5 关于flatMap()方法

在程序设计中有时会出现巢状或瀑布式的流程，就结构来看每一层运算极为类似，只是返回的类型不同，很难抽取流程重用。举例来说，如果你的方法可能返回null，你可能会设计出某个流程如下：

```
Customer customer = order.getCustomer();
if(customer != null) {
    String address = customer.getAddress();
    if(address != null) {
        return address;
    }
}
return "n.a.";
```

巢状的层次可能还会更深，例如：

```
Customer customer = order.getCustomer();
if(customer != null) {
    Address address = customer.getAddress();
    if(address != null) {
        City city = address.getCity();
        if(city != null) {
            ...
        }
    }
}
return "n.a.";
```

严禁网络传播本PDF，违者责任自负！

在连续的层次不深时，也许程序代码看来还算直观，然而层次变深之后，显然地，很容易迷失在层次之中，虽然每层都是判断值是否为 null，不过因为类型不同，看来不太好抽取流程重用。

就 12.2.1 节中的介绍，方法可能有或没有值时，不建议使用 null 作为没有值的代表。如果能修改 getCustomer()返回 Optional<Customer>，也修改 getAddress()返回 Optional<String>，那一开始的程序片段可以改为：

```
String addr = "n.a.";
Optional<Customer> customer = order.getCustomer();
if(customer.isPresent()) {
    Optional<String> address = customer.get().getAddress();
    if(address.isPresent()) {
        addr = address.get();
    }
}
return addr;
```

看来好像没有高明到哪去，不过至少每一层都是 Optional 类型了，而每一层都是 isPresent()的判断，然后将 Optional<T>转换为 Optional<U>，如果将 Optional<T>转换为 Optional<U>的方式可以由外部指定，那你就可以重用 isPresent()的判断了，实际上 Optional 有个 flatMap()方法，已经帮你写好这个逻辑了：

```
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent())
        return empty();
    else {
        return Objects.requireNonNull(mapper.apply(value));
    }
}
```

所以，可以直接使用 Optional 的 **flatMap()**方法：

```
return order.getCustomer()
            .flatMap(Customer::getAddress)
            .orElse("n.a.");
```

如果层次不深，也许看不出使用这个的好处，若层次变深可以发现其益处，如一开始第二个程序片段，像下面这样改写就清楚多了：

```
return order.getCustomer()
            .flatMap(Customer::getAddress)
            .flatMap(Address::getCity)
            .orElse("n.a.");
```

Optional 的 flatMap()这个名称令人困惑，可从 Optional<T>调用 flatMap()后得到 Optional<U>来想象一下，**flatMap()**就像是从盒子取出另一盒子(flat 就是平坦化的意思)，Lambda 表达式指定了前一个盒子中的值与下一个盒子之间的转换关系，因为判断是否有值的运算情境被隐藏了，因此用户可明确指定感兴趣的特定运算，从而使程序代码意图显露出来，又可顺畅地连续运算，以避免巢状或瀑布式的复杂检查流程。

如果没办法修改 getCustomer()、getAddress()、getCity()等返回 Optional 类型怎么办？Optional 还有个 **map()**方法。例如，参数 order 是 Order 类型，有 null 的可能性，getCustomer()、

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

getAddress()、getCity()等的返回类型分别是Customer、Address、City，且有可能返回null，那么就可以这么做：

```
return Optional.ofNullable(order)
               .map(Order::getCustomer)
               .map(Customer::getAddress)
               .map(Address::getCity)
               .orElse("n.a.");
```

与flatMap()的差别在于，map()方法操作中，对mapper.apply(value)的结果使用了Optional.ofNullable()方法(flatMap()中使用的是Objects.requireNonNull())，因此有办法持续处理null的情况：

```
public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent())
        return empty();
    else {
        return Optional.ofNullable(mapper.apply(value));
    }
}
```

如果之前的Order有个getLineItems()方法，可取得订单中的产品项目List<LineItem>，想要取得LineItem的名称，可以通过getName()来取得，若你有个List<Order>，想取得所有的产品项目名称会怎么写？直观的写法应该是用循环：

```
List<String> itemNames = new ArrayList<>();
for(Order order : orders) {
    for(LineItem lineItem : order.getLineItems()) {
        itemNames.add(lineItem.getName());
    }
}
```

当然，层次不深时这样写很直觉，也算容易阅读，不过层次深时，例如，想进一步取得LineItem的赠品名称的话，又得多一层for循环，如果还要继续取下去呢？

可以在用List的stream()方法取得Stream之后，使用flatMap()方法如下改写：

```
List<String> itemNames = orders.stream()
        .map(Order::getLineItems)
        .flatMap(lineItems -> lineItems.stream())
        .map(LineItem::getName)
        .collect(toList());
```

就程序代码阅读来说，第一个stream()方法返回Stream<Order>，紧接着的map()返回Stream<List<LineItem>>，如果将Stream看作是一个盒子，List<LineItem>就是目前盒中的值，flatMap()指定的Lambda运算，指定了目前盒中的List<LineItem>值与下一个盒子Stream<LineItem>间的关系。

提示»» 刚才的程序片段，flatMap(lineItems -> lineItems.stream())该行，也可以使用方法参考的方式flatMap(List::stream)。

这类从盒子中取出盒子的操作(一个Stream接着一个Stream)可以连续下去。例如，想进一步取得LineItem的赠品名称可以如下：

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
List<String> itemNames = orders.stream()
        .map(Order::getLineItems)
        .flatMap(lineItems -> lineItems.stream())
        .map(LineItem::getPremiums)
        .flatMap(premiums -> premiums.stream())
        .map(Premium::getName)
        .collect(toList());
```

提示>>> 类似地，flatMap(lineItems -> lineItems.stream())或 flatMap(premiums -> premiums.stream())该行，都可以改用 flatMap(List::stream)。

基本上，如果能了解 Optinal、Stream(或其他类型)的 flatMap()方法，其实就是取得盒子中的值，指定这个值与下个盒子间的关系，在撰写与阅读程序代码时，忽略掉 flatMap 这个名称，就能清楚程序代码的主要意图。

提示>>> flatMap()方法的概念，其实来自于函数程序设计(Functional Programming)中的单子(Monad)概念，有兴趣进一步了解的话，可以从以下文档开始：
http://www.slideshare.net/mariofusco/monadic-java

在 JDK9 中，Collectors 上新增了 flatMapping()方法，可指定 flatMap 操作并返回 Collector 实例。flatMapping()方法用来减少管道操作的层次，或者建立可重用的 Collector，例如原先有个操作：

```
List<String> addressLt = customers.stream()
        .flatMap(customer -> customer.getAddressList().stream())
        .collect(toList());
```

若改用 flatMapping()方法，可以如下：

```
List<String> addressLt = customers.stream()
     .collect(flatMapping(
         customer -> customer.getAddressList().stream(), toList())
      );
```

12.2.6 Stream 相关 API

在许多 API 上，都可以取得 Stream 实例。举例来说，如果想对数组进行管道化操作，方法之一是使用 Arrays 的 asList()或 List 的 of()方法返回 List，而后调用 stream()方法获取 Stream 实例；另一个方法是使用 Arrays 的 **stream()**方法，它可以指定数组后返回 Stream 实例。

实际上，Stream、IntStream、DoubleStream 等都有 **of()**静态方法，可以使用可变长度自变量方式指定元素，各可返回 Stream、IntStream、DoubleStream 实例，它们也各有 **generate()**与 **iterate()**静态方法，可以分别建立 Stream、IntStream、DoubleStream 实例。

如果你想要产生一个整数范围，IntStream 上有 **range()**与 **rangeClosed()**方法，它们返回 IntStream 实例。range()与 rangeClosed()方法的差别在于，后者返回的范围会包括第二个参数指定的值。例如，如果你原先这么撰写：

```
for (int i = 0 ; i < 10000 ; i++) {
  out.println(i);
}
```

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

现在可以改使用 range() 如下撰写：

```
range(0, 10000).forEach(out::println);
```

JDK8 的 CharSequence 上增加了 **chars()** 与 **codePoints()** 两个方法，都是返回 IntStream，前者代表一串字符的整数值，后者代表一串字符的码点（Code point）。例如：

```
IntStream charStream = "Justin".chars();
IntStream codeStream = "Justin".codePoints();
```

如果你需要产生随机数，在 java.lang.Math 上有个 random() 静态方法，不过使用上并不是那么方便，JDK8 中增加了个实用的 java.util.Random 类，来看一下实际的例子：

```
Random random = new Random();
DoubleStream doubleStream = random.doubles(); // 0 到 1 间的随机浮点数
IntStream intStream = random.ints(0, 100);    // 0 到 100 间的随机整数
```

提示 »» Math 在 JDK8 中也增加了很多数学运算相关 API，例如 multiplyExact()、floorMod()、floorDiv()等，详情可参考 API 文件。

12.2.7 JDK9 Optional 与 Stream 增强

在 JDK9 中，为 Optional 与 Stream 做了一些 API 增强。如果你对于 Optional 与 Stream 的使用已经得心应手，可以考虑采用这些新的 API 让程序的撰写更为便捷。

1. Optional 的 API 增强

来看一个程序片段，如果 getNickName() 会返回 Optional<String>：

```
Optional<String> nickOptional = getNickName("Duke");
if(nickOptional.isPresent()) {
    String nickName = nickOptional.get();
    out.printf("Hello, %s%n", nickName);
} else {
    out.println("Hello, Guest");
}
```

类似的流程，你可能撰写过许多次了，在 JDK9 中，现在可以直接写为：

```
getNickName("Duke").ifPresentOrElse(
    nickName -> out.printf("Hello, %s%n", nickName),
    () -> out.println("Hello, Guest")
);
```

如果返回的 Optional 实例调用 isPresent() 返回 true，会执行第一个 Lambda 表示式，否则执行第二个 Lambda 表示式。

随着 Optional 在越来越多地方使用，就会开始考虑到 API 衔接的问题。例如，若 getNickName() 与 getDefaultName() 都返回 Optional<String>，而你曾经定义过这类的方法：

```
public Optional<String> getDisplayName(String username) {
    Optional<String> nickOptional = getNickName(username);
    if(nickOptional.isPresent()) {
       return nickOptional;
    } else {
       return getDefaultName(username);
    }
}
```

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

像这类需求，在 JDK9 中可以改写如下：

```
public Optional<String> getDisplayName(String username) {
    return getNickName(username).or(
        () -> getDefaultName(username)
    );
}
```

由于某些原因，你可能会使用一组名称来调用 getNickName()，例如：

```
public Stream<String> getAvailableNickNames(List<String> usernames) {
    return usernames.stream()
                    .map(username -> getNickName(username))
                    .filter(opt -> opt.isPresent())
                    .map(opt -> opt.get());
}
```

getAvailableNickNames() 会用来获取用户的昵称清单，由于 getNickName() 返回 Optional<String>，因此 map(username -> getNickName(username)) 返回类型会是 Stream<Optional<String>>，因此 filter(opt -> opt.isPresent()) 就只留下有设定的昵称，返回类型也是 Stream<Optional<String>>。然而 getAvailableNickNames() 最后必须返回 Stream<String>，因此再使用一次 map(opt -> opt.get()) 返回 Stream<String>。

如果是 JDK9 的话，可以直接改用 Optional 的 stream() 方法，并搭配 Stream 的 flatMap() 来简化程序的撰写，下面的片段也使用了方法参考让程序更简洁：

```
public Stream<String> getAvailableNickNames(List<String> usernames) {
    return usernames.stream()
                    .map(this::getNickName))
                    .flatMap(Optional::stream);
}
```

map(this::getNickName)) 返回 Stream<Optional<String>>，记得吗？Stream<T> 的 flatMap() 会从 Stream<T> 中逐一取得内含的 T，就上例而言就是从 Stream<Optional<String>> 取得 Optional<String>，而 flatMap() 接受的 Lambda 表示式必须返回 Stream<T>。就上面的例子而言，Lambda 表示式必须返回 Stream<Optional<String>>，而 Optional<String> 的 stream() 方法就是返回 Stream<Optional<String>> 类型，而 Optional 的 stream() 中也做了 isPresent() 的检查。

2. Stream 的 API 增强

如果 order 有个 getCustomer() 方法返回类型为 Customer，你想要在返回的 Customer 实例上调用 getEmailList() 取得联系用的邮件清单，然而 getCustomer() 可能返回 null，因此你也许撰写出以下的程序：

```
Customer customer = order.getCustomer();
Stream<String> emailLt = customer == null
     ? Stream.empty()
     : customer. getEmailList().stream();
```

OK！因为你已经很熟悉 Optional 的运用了，看到 null 检查，你会想要修改为以下的版本：

```
Stream<String> emailLt =
     Optional.of(order.getCustomer())
             .map(cust -> cust.getEmailList().stream())
             .orElse(Stream.empty());
```


严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

如果在 JDK9 的话，可以使用 Stream.ofNullable()来进一步简化程序：

```
Stream<String> emailLt =
    Sream.ofNullable(order.getCustomer())
        .flatMap(cust -> cust.getEmailList().stream());
```

Stream 有个接受单一自变量的版本 of()，若传入 null 会抛出 NullPointerException，而 Stream.ofNullable()若接受 null，会返回一个空的 Stream（调用 Stream.empty()），否则使用 Stream.of()建立一个含有单一元素的 Stream 实例，如此一来就不用通过 Optional.ofNullable()来衔接 API 了。

在 JDK8 中的 Stream 只有一个 iterate()版本，如果你试着运行下面的程序，那么它会从 x 为 4 开始不断地迭代：

```
Stream.iterate(4, x -> x + 1).forEach(out::println);
```

4 是 x 的初始值，Lambda 表示式的运算结果会成为下个 x 值，因而你会看到数字不断地递增显示，如果你想要设定迭代中止的条件，在 JDK9 中新增了另一个 iterate()版本：

```
jshell> import static java.lang.System.out;

jshell> Stream.iterate(4, x -> x < 10, x -> x + 1).forEach(out::println);
4
5
6
7
8
9
```

JDK9 还新增了 takeWhile()与 dropWhile()方法，takeWhile()保留符合条件的元素，直到遇到第一个不符合的元素就终止；dropWhile()则是相反，会丢弃条件的元素直到遇到第一个不符合的元素，例如：

```
jshell> Stream.of(1, 2, 3, 2, 1).takeWhile(x -> x < 3).forEach(out::println);
1
2

jshell> Stream.of(1, 2, 3, 2, 1).dropWhile(x -> x < 3).forEach(out::println);
3
2
```

12.3 Lambda、平行化与异步处理

引入 Lambda 的目的之一，是让开发者在撰写平行程序时更为简便。然而想要获得便利性的前题是，开发者在设计上必须先有分而治之的概念。在这一节中，简介利用 Lambda 时必须有哪些考虑，才能在有平行设计需求时，轻松拥有并行处理的能力。

12.3.1 Stream 与平行化

在 12.2.4 节中提到“Collector 的 accumulator()的作用，在使用具有并行处理能力的 Stream

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

时……”，这表示 Stream 有办法进行并行处理？是的，只要设计适当，想要获得并行处理能力可以说很简单，例如这段程序代码：

```
List<Person> males = persons.stream()
                .filter(person -> person.getGender() == Person.Gender.MALE)
                .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

只要将 stream()改成 **parallelStream()**，就可能拥有并行处理之能力：

```
List<Person> males = persons.parallelStream()
                .filter(person -> person.getGender() == Person.Gender.MALE)
                .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

Collection 的 parallelStream()方法，返回的 Stream 实例在操作时，会在可能的情况下进行并行处理，Java 在 API 的设计上，希望你想要进行并行处理时，必须有明确的语义，这也是为什么会有 stream()与 parallelStream()两个方法，前者代表循序(Serial)处理。后者代表并行处理，想要知道 Stream 是否为并行处理，可以调用 **isParallel()**来得知。

只不过，也并不是任何用到 stream()方法的程序代码，只要改成了 parallelStream()，就可以拥有并行处理能力，毕竟天下没有白吃的午餐，你还是得留意一些设计上的问题。

1. 留意并行处理时的顺序需求

使用了 **parallelStream()**，不代表一定会并行处理而使得执行必然变快。要调用哪个方法，必须思考你的处理过程是否能够分而治之然后合并结果，在这个例子中，filter()与 collect()方法基本上都有可能。

类似地，Collectors 有 groupingBy()与 **groupingByConcurrent()**两个方法，前者代表循序处理，后者代表并行处理。是否调用后者，同样你得思考处理过程是否能够分而治之然后合并结果，如果可能，才能从中获益。例如原先有段程序：

```
Map<Person.Gender, List<Person>> males = persons.stream()
                .collect(
                    groupingBy(Person::getGender));
```

想要在可能的情况下进行并行处理，可以改为：

```
Map<Person.Gender, List<Person>> males = persons.parallelStream()
                .collect(
                    groupingByConcurrent(Person::getGender));
```

Stream 实例若具有并行处理能力，处理过程会分而治之，也就是将任务切割为小任务，这表示每个小任务都是一个管道化操作，因此像以下的程序片段：

```
jshell> import static java.lang.System.out;

jshell> List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
numbers ==> [1, 2, 3, 4, 5, 6, 7, 8, 9]

jshell> numbers.parallelStream().forEach(out::println);
6
5
8
9
7
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
3
4
1
2
```

得到的显示顺序不一定会是1、2、3、4、5、6、7、8、9，而可能是任意的顺序，就forEach()这个终结操作来说，如果在并行处理时，希望最后顺序是照着原来Stream来源的顺序，那可以调用**forEachOrdered()**。例如：

```
jshell> numbers.parallelStream().forEachOrdered(out::println);
1
2
3
4
5
6
7
8
9
```

在管道操作时，如果forEachOrdered()中间有其他如filter()的中间操作，会试着平行化处理，然而最终forEachOrdered()会以来源顺序处理。因此，使用**forEachOrdered()**这类的有序处理时，可能会(或完全失去)失去并行化的一些优势，实际上中间操作也有可能如此。例如**sorted()**方法。

在使用Stream的reduce()与collect()时，并行处理时也得留意一下顺序，API文件上基本上会记载终结操作时是否按来源顺序，reduce()基本上是按照来源顺序，而collect()得视给予的Collector而定，在以下两个例子中collect()都是按照来源顺序处理：

```
List<Person> males = persons.parallelStream()
                   .filter(person -> person.getGender() == Gender.MALE)
                   .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);

List<Person> males = persons.parallelStream()
                   .filter(person -> person.getGender() == Gender.MALE)
                   .collect(toList());
```

在collect()操作时若想要有平行效果，必须符合以下三个条件：

- Stream有并行处理能力。
- Collector有Collector.Characteristics.CONCURRENT特性。
- Stream是无序的(Unordered)或者是Collector具有Collector.Characteristics.UNORDERED特性。

想要知道Collector是否具备Collector.Characteristics.UNORDERED或Collector.Characteristics.CONCURRENT特性，可以调用Collector的**characteristics()**方法，并行处理的Stream基本上是无序的，如果不放心，可以调用Stream的**unordered()**方法。

Colllector具有CONCURRENT与UNORDERED特性的例子之一是Collectors的**groupingByConcurrent()**方法返回的实例，因此在最后顺序不重要时，使用groupingByConcurrent()来取代groupingBy()方法，在效能上会有所帮助。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

2. 不要干扰 Stream 来源

想要用好 JDK 提供的并行处理功能，数据处理过程必须能够分而治之，然后将每个小任务的结果加以合并，这表示当 API 在处理小任务时，不应该进行干扰，例如：

```
numbers.parallelStream()
       .filter(number -> {
            numbers.add(7);
            return number > 5;
       })
       .forEachOrdered(out::println);
```

无论是基于哪种理由，像这类对源数据的干扰都令人困惑，实际上无论是否进行并行处理，这样的程序都会引发 ConcurrentModifiedException。

3. 一次做一件事

JDK 提供高级语义的管道化 API，在可能的情况下实现惰性、并行处理能力，目的之一是希望你思考处理的过程中，实际上是由哪些小任务组成。在过去，你可能基于(自我想象的)效率增进考虑，在循环中做了很多件事，因而让程序变得复杂。现在使用了高级 API，就要避免走回头路。例如，过去你在写 for 循环时，可能会顺便做些动作，如过滤元素做显示的同时，将元素做个运算并收集在另一个清单中：

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> alsoLt = new ArrayList<>();

for(Integer number : numbers) {
    if(number > 5) {
        alsoLt.add(number + 10);
        out.println(number);
    }
}
```

在 JDK 中使用高级 API 重构(Refactor)以上程序代码时，记得一次只做一件事：

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

List<Integer> biggerThan5 = numbers.stream()
                .filter(number -> number > 5)
                .collect(toList());

biggerThan5.forEach(out::println);

List<Integer> alsoLt = biggerThan5.stream()
               .map(number -> number + 10)
               .collect(toList());
```

避免写出以下的程序：

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> alsoLt = new ArrayList<>();

numbers.stream()
       .filter(number -> {
           boolean isBiggerThan5 = number > 5;
           if(isBiggerThan5) {
               alsoLt.add(number + 10);
           }
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
            return isBiggerThan5;
        })
        .forEach(out::println);
```

这样的程序不仅不易理解，如果试图进行平行化处理时：

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> alsoLt = new ArrayList<>();

numbers.parallelStream()
        .filter(number -> {
            boolean isBiggerThan5 = number > 5;
            if(isBiggerThan5) {
                alsoLt.add(number + 10);
            }
            return isBiggerThan5;
        })
        .forEachOrdered(out::println);
```

就会发现，alsoLt 的顺序并不按照 numbers 的顺序，然而前面一次处理一个任务的版本，可以简单地改为平行化版本，又没有顺序问题：

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

List<Integer> biggerThan5 = numbers.parallelStream()
                .filter(number -> number > 5)
                .collect(toList());

biggerThan5.forEach(out::println);

List<Integer> alsoLt = biggerThan5.parallelStream()
                .map(number -> number + 10)
                .collect(toList());
```

12.3.2 Arrays 与平行化

从 JDK8 开始，针对大型数组的平行化操作，在 Arrays 上增加了 **parallelPrefix()**、**parallelSetAll()**与 **parallelSort()**方法，它们都有多个重载版本。

parallelPrefix()方法可以指定 XXXBinaryOperator 实例，执行类似 Stream 的 reduce()过程，也就是 XXXBinaryOperator 的 applyXXX()方法第一个参数接受前前次运算结果，第二个参数接受数组目前代入的元素。例如：

```
int[] arrs = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(arrs, (left, right) -> left + right);
out.println(Arrays.toString(arrs)); // [1, 3, 6, 10, 15]
```

parallelSetAll()用来对数组进行初始化或全面重设每个索引元素，可指定 XXXFunction 或 IntUnaryOperator，每次会代入索引值，你指定目前索引位置该设定的元素。例如：

```
int[] arrs = new int[10000000];
Arrays.parallelSetAll(arrs, index -> -1);
```

parallelSort()方法，可以将指定的数组分为子数组并以平行化方式分别排序，然后再进行合并排序。指定的数组之元素必须操作 Comparable，或是可以对 parallelSort()指定 Comparator。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

12.3.3 CompletableFuture 非同步处理

如果想要异步(Asyncronous)读取文本文件，在文档读取完后做某些事，利用 11.2.2 节的内容，可以使用 ExecutorService 来 submit() 一个 Runnable 对象。如类似以下的流程：

```
public static Future readFileAsync(String file, Consumer<String> success,
                    Consumer<IOException> fail, ExecutorService service) {
    return service.submit(() -> {
        try {
            success.accept(new String(Files.readAllBytes(Paths.get(file))));
        } catch (IOException ex) {
            fail.accept(ex);
        }
    });
}
```

这么一来，就可使用以下异步的风格来读取一个文本文件：

```
readFileAsync(args[0],
    content -> out.println(content),        // 成功处理
    ex -> ex.printStackTrace(),             // 失败处理
    Executors.newFixedThreadPool(10)
);
```

out.println(content) 与 ex.printStackTrace() 会在与读取文档的同一个线程中进行，如果想要在不同线程中进行，得再额外做些设计；此外，这种异步操作使用的回调(Callback)风格，在每次回调中若又再度进行异步操作及回调，很容易写出回调地狱(Callback hell)，造成可读性不佳。例如有个类似 readFileAsync() 风格的异步 processContentAsync() 方法，用来再继续处理 readFileAsync() 读取的档案内容，那么可能撰写出以下的程序代码：

```
readFileAsync(args[0],
    content -> processContentAsync(content,
                processedContent -> out.println(processedContent) ,
                ex -> ex.printStackTrace(), service),
    ex -> ex.printStackTrace(), service);
```

实际上异步处理的组合需求很多，可以使用 **java.util.concurrent.CompletableFuture** 来改写 readFileAsync()，例如：

Lambda Async.java

```
package cc.openhome;

import java.io.*;
import static java.lang.System.out;
import java.nio.file.*;
import java.util.concurrent.*;

public class Async {
    public static CompletableFuture<String> readFileAsync(
                    String file, ExecutorService service) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                return new String(Files.readAllBytes(Paths.get(file)));
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
                } catch(IOException ex) {
                    throw new UncheckedException(ex);
                }
            }, service);
        }

        public static void main(String[] args) throws Exception {
            ExecutorService poolService = Executors.newFixedThreadPool(10);

            readFileAsync(args[0], poolService).whenComplete((ok, ex) -> {
                Optional.ofNullable(ex)
                        .ifPresentOrElse(
                            Throwable::printStackTrace,
                            () -> out.println(ok)
                            );
            }).join(); // join()是为了避免main线程在任务完成前就关闭ExecutorService

            poolService.shutdown();
        }
}
```

CompletableFuture 的静态方法 **supplyAsync()**接受 Suppiier 实例，可指定异步执行任务，在上面的范例中会由指定的 Executor 中某个线程来执行。supplyAsync()另一个版本不用指定 Executor 实例，异步任务将由 ForkJoinPool.commonPool()返回的 ForkJoinPool 中某个线程来执行。

supplyAsync()会返回 CompletableFuture 实例，可以调用 **whenComplete()**以 BiConsumer 实例指定任务完成如何处理，第一个参数是 Suppiier 的返回值，若有例外发生则会指定给第二个参数，范例中还使用了 JDK9 中 Optional 新增的 ifPresentOrElse()方法；想要在任务完成后继续异步地处理，则可以使用 **whenCompleteAsync()**方法。

如果第一个 CompletableFuture 任务完成后，想要继续以异步方式来处理结果，可以使用 **thenApplyAsync()**。例如：

```
readFileAsync(args[0], poolService)
    .thenApplyAsync(String::toUpperCase)
    .whenComplete((ok, ex) -> {
        Optional.ofNullable(ex)
                .ifPresentOrElse(
                    Throwable::printStackTrace,
                    () -> out.println(ok)
                );
    });
```

CompletableFuture 实例的方法，基本上都会有同步与异步两个版本，可以用 Async 后置名称来区分，例如，thenApplyAsync()的同步版本就是 **thenApply()**方法。

之前介绍过 Optional 与 Stream 中各定义有 map()方法，可让你指定 Optional 或 Stream 中的值 T 如何映射为值 U，然后返回新的 Optional 或 Stream，**CompletableFuture** 的 **thenApply()**(以及异步的 **thenApply()**版本)其实就类似 **Optional** 或 **Stream** 的 **map()**，可让你指定前一个 CompletableFuture 处理后的结果 T 如何映射为值 U，然后返回新的 CompletableFuture。

之前也介绍过 Optional 与 Stream 中也各定义有 flatMap()方法，可指定 Optional 或 Stream 中的值 T 与 Optional<U>、Stream<U>之间的关系，**CompletableFuture** 也有个 **thenCompose()**(以及异

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

步的 **thenComposeAsnyc()**版本)，作用就类似 **flatMap()**，可以指定前一个 CompletableFuture 处理后的结果 T 如何映像为值 CompleteableFuture<U>，举例来说，你想在 readFileAsync()返回的 CompletableFuture <String> 处理完后，继续组合 processContentAsync() 方法返回 Completable-Future<String>，就可以如下撰写：

```
readFileAsync(args[0], poolService)
        .thenCompose(content -> processContentAsync(content, poolService))
        .whenComplete((ok, ex) -> {
            Optional.ofNullable(ex)
                    .ifPresentOrElse(
                        Throwable::printStackTrace,
                        () -> out.println(ok)
                    );
        });
```

提示》》 CompletableFuture 上还有许多方法可以使用，详情除了参考 API 文件之中，读者还可以看看以下文章 *Java 8: Definitive guide to CompletableFuture* 作为开始：http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html

12.3.4 JDK9 CompletableFuture 增强

在 JDK9 中，对 CompletableFuture 也做了一些 API 增强。例如，若想要延迟执行的话，可以使用 static 的 delayedExecutor()方法：

```
public static Executor delayedExecutor(
    long delay, TimeUnit unit)
public static Executor delayedExecutor(
    long delay, TimeUnit unit, Executor executor)
```

两个方法都会返回 Executor 实例，第一个方法会在指定的时间延迟之后，才将任务发送给默认的 Executor，而第二个方法会在指定的时间延迟之后，才将任务发送给指定的 Executor。TimeUnit 是位于 java.util.concurrent.TimeUnit 的枚举类型，包含 SECONDS、MINUTES 等枚举成员。例如：

```
...
future.completeAsync(
     () -> "Orz",
     CompletableFuture.delayedExecutor(3, TimeUnit.SECONDS)
   )
   .whenComplete((ok, ex) -> out.println(ok));
...
```

在这个程序片段中还示范了 JDK9 新增的 completeAsync()方法（它也有另一个使用默认 Executor 的版本），这个方法第一个 Suppiler 自变量的返回值，将会做为 CompletableFuture 任务执行结果。由于使用了 delayedExecutor()，上面的程序片段中，completeAsync()会在延迟 3 秒之后才完成任务。

JDK9 中 CompletableFuture 新增了 orTimeout()方法，当任务执行超过指定的时间，会抛出 java.util.concurrent.TimeoutException：

```
public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

例如，前面的 Async 范例中，如果想在读取时间超过 3 秒之后抛出 TimeException，可以修改如下：

```
...
    readFileAsync(args[0], poolService)
       .orTimeOut(3, TimeUnit.SECONDS)
       .whenComplete((ok, ex) -> {
          Optional.ofNullable(ex)
                  .ifPresentOrElse(
                     Throwable::printStackTrace,
                     () -> out.println(ok)
                  );
    }).join();
...
```

也可以使用 completeOnTimeOut()指定超时发生时的任务完成结果：

```
...
    readFileAsync(args[0], poolService)
        .completeOnTimeOut("TimeOut Happens", 3, TimeUnit.SECONDS)
        .whenComplete((ok, ex) -> {
           Optional.ofNullable(ex)
                   .ifPresentOrElse(
                      Throwable::printStackTrace,
                      () -> out.println(ok)
                   );
    }).join();
...
```

提示 »» 除了这里介绍的 API 增强之外，CompletableFuture 上还有一些新增的方法，详情可参考 *Java 9 CompletableFuture API Improvements*，网址如下：

www.baeldung.com/java-9-completablefuture

异步处理的领域中，还发展出 Reactive Programming 典范，作为新特性，JDK9 也引入了 Flow API 来支持 Reactive。然而只是单纯想在 JDK 中标准化(以及简单地提供了 SubmissionPublisher 基础类)，想要以 Reactive 典范进行设计，Java 社群仍主要采用第三方程式库(像是 RxJava 2.0)，Flow API 在 JDK9 中没有实质的作用。

若是对于 Reactive 典范与设计若有兴趣，可以从《Reactive 与 Java 9》这篇文件作为开始，网址如下：

openhome.cc/Gossip/Programmer/ReactiveJa

12.4 重点复习

在 Java 中引入 Lambda 的同时，与现有 API 维持兼容性是主要考虑之一。方法参考的特性，在重用现有 API 上扮演了重要角色。重用现有方法操作，可避免到处写下 Lambda 表达式。方法参考不仅避免了重复撰写 Lambda 表达式，也可以让程序代码更为清楚。

在只有 Lambda 表达式的情况下，参数的类型必须写出来，如果有目标类型的话，在编译程序可推断出类型的情况下，就可以不写出 Lambda 表达式的参数类型。Lambda 表达式本身是中性的，不代表任何类型的实例。同样的 Lambda 表达式，可用来表示不同目标类型的对象操作。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Lambda 并没有导入新类型来作为 Lambda 表达式的类型，而是就现有的 interface 语法来定义函数接口，作为 Lambda 表达式的目标类型。函数接口就是接口，但要求仅具单一抽象方法，许多现存的接口都是这种接口，例如标准 API 中的 `Runnable`、`Callable`、`Comparator` 等。

匿名类不是不好，只不过有其应用的场合，在许多时候，特别是接口只有一个方法要操作时，你会只想关心参数及操作本身，不想理会类与方法名称，Lambda 表达式只关心方法命名上的参数与返回定义，但忽略方法名称。

JDK8 对 interface 语法做了改进，允许有默认方法操作。

Lambda 表达式并不是匿名类的语法蜜糖。视 Lambda 表达式是在哪个名称范畴，就能参考该范畴内的名称，如变量或方法。JDK8 之前，如果要在匿名内部类中存取局部变量，则该局部变量必须是 `final`，否则会发生编译错误。而在 JDK8 之后，如果变量本身等效于 `final` 局部变量，也就是说，如果变量不会在匿名类中有重新指定的动作，就可以不加上 `final` 关键词。如果 Lambda 表达式中捕获的局部变量本身等效于 `final` 局部变量，可以不在局部变量上加上 `final`，JDK 禁止用户在 Lambda 中修改局部变量的值。

只要静态方法的方法命名中参数与返回值定义相同，也可以使用静态方法来定义函数接口操作。

除了参考静态方法作为函数接口操作之外，还可以参考特定对象的实例方法。函数接口操作也可以参考类上定义的非静态方法，函数接口会试图用第一个参数方法接收者，而之后的参数依次作为被参考的非静态方法的参数。JDK 还提供了构造函数参考，用来重用现有 API 的对象建构流程。

默认方法必须使用 `default` 关键词修饰，默认权限为 `public`，不能使用数据成员。从 JDK9 开始，接口中也可以定义 `private` 方法了。

JDK8 除了让接口可以定义默认方法之外，也开始允许在接口中定义静态方法，从 JDK9 开始，也可以定义为 `private` 的静态方法了。

调用方法时如果返回类型是 `Optional`，应该立即想到它可能包含也可能不包含值。通用函数接口基本上放置于 `java.util.function` 套件之中，就行为来说，基本上可以分为 `Consumer`、`Function`、`Predicate` 与 `Supplier` 四个类型。

JDK8 引入了 `Stream` API，也引入了管道操作风格，一个管道包括了几个部分：来源、零或多个中间操作、一个最终操作。

12.5 课后练习

请使用 9.1.5 节中定义的 `ArrayList`，在其上增加 `filter()`、`map()`、`reduce()` 与 `forEach()` 方法，使得 `ArrayList` 实例可以进行如下操作：

```
ArrayList<Integer> numbers = new ArrayList<>();
...使用 add() 新增 Integer 元素
numbers.filter(n -> n > 5).forEach(out::println);
numbers.map(n -> n * 2).forEach(out::println);
out.println(numbers.reduce((total, n) -> total + n).orElse(0));
```

Chapter 13 时间与日期

学习目标

- 建立时间与日期
- 认识 Date 与 Calendar
- 使用时间日期 API
- 区分机器与人类时间概念

13.1 认识时间与日期

在正式认识 Java 提供了哪些时间处理 API 之前，得先来了解一些时间、日期的历史问题，这样你才会知道，时间日期确实是个很复杂的问题，而使用程序来处理时间日期，也不仅仅是使用 API 的问题。

13.1.1 时间的度量

想度量时间，得先有个时间基准，大多数人知道格林威治(Greenwich)时间，那么就先从这个时间基准开始认识。

1. 格林威治标准时间

格林威治标准时间(Greenwich Mean Time)，经常简称 GMT，一开始是参考自格林威治皇家天文台的标准太阳时间，格林威治标准时间的正午是太阳抵达天空最高点之时，GMT 时间常被不严谨(且有争议性)地当成是世界协调时间(UTC)。

GMT 通过观察太阳而得，然而地球公转轨道为椭圆形且速度不一，本身自转亦缓慢减速中，因而会造成越来越大的时间误差，现在 GMT 已不作为标准时间使用。

2. 世界时

世界时(Universal Time，UT)是借由观测远方星体跨过子午线(Meridian)而得，这会比观察太阳来得准确一些。公元 1935 年，International Astronomical Union 建议使用更精确的 UT 来取代 GMT，在 1972 年引入 UTC 之前，GMT 与 UT 是相同的。

3. 国际原子时

虽然观察远方星体会比观察太阳来得精确，不过 UT 基本上仍受地球自转速度影响而会有误差。1967 年定义的国际原子时(International Atomic Time，TAI)，将秒的国际单位(International System of Units，SI)定义为铯(Caesium)原子辐射振动 9192631770 周耗费的时间，时间从 UT 的 1958 年开始同步。

4. 世界协调时间

由于基于铯原子振动定义的秒长是固定的，然而地球自转会越来越慢，这会使得实际上 TAI 时间会不断超前基于地球自转的 UT 系列时间，为了保持 TAI 与 UT 时间不要差距过大，因而提出了具有折中修正版本的世界协调时间(Coordinated Universal Time)，常简称为 UTC。

UTC 经过了几次的时间修正，为了简化日后对时间的修正，1972 年 UTC 采用了闰秒(Leap Second)修正(1 January 1972 00:00:00 UTC 实际上为 1 January 1972 00:00:10 TAI)，确保 UTC 与 UT 相差不会超过 0.9 秒，加入闰秒的时间通常会在 6 月底或 12 月底，由巴黎的 International Earth Rotation and Reference Systems Service 负责决定何时加入闰秒。

最近一次的闰秒修正为 2016 年 12 月 31 日，当时 TAI 实际上已超前 UTC 有 37 秒之长。

5. UNIX 时间

Unix 系统的时间表示法，定义为 UTC 时间 1970 年(Unix 元年)1 月 1 日 00:00:00 为起点而经过的秒数，不考虑闰秒修正，用以表达时间轴上某一瞬间(Instant)。

6. epoch

某个特定时代的开始，时间轴上某一瞬间。例如 Unix epoch 选为 UTC 时间 1970 年 1 月 1 日 00:00:00，不少来自于 Unix 的系统、平台、软件等，也都选择这个时间作为时间表示法的起算点，例如稍后要介绍的 `java.util.Date` 封装的时间信息，就是 January 1, 1970, 00:00:00 GMT(实际上是 UTC)经过的毫秒数，可以简称它为 epoch 毫秒数。

提示 »» 以上是关于时间日期的重要整理，为了帮助读者后续 API 知道该如何使用。有机会的话，应该在维基百科上详细认识时间与日期：
http://zh.wikipedia.org/

就以上这些介绍来说有几个重点：

- 就目前来说，即使标注为 GMT(无论是文件说明，或者是 API 的日期时间字符串描述)，实际上谈到时间指的是 UTC。
- 秒的单位定义是基于 TAI，也就是铯原子辐射振动次数。
- UTC 考虑了地球自转越来越慢而有闰秒修正，确保 UTC 与 UT 相差不会超过 0.9 秒。最近一次的闰秒修正为 2016 年 12 月 1 日，当时 TAI 实际上已超前 UTC 有 37 秒之长。
- Unix 时间是 1970 年 1 月 1 日 00:00:00 为起点而经过的秒数，不考虑闰秒，不少来自于 Unix 的系统、平台、软件等，也都选择这个时间作为时间表示法的起算点。

13.1.2 年历简介

度量时间是一回事，表达日期又是另一回事，前面讲到的时间起点都是使用公历，中文国人习惯称为阳历。在介绍公历之前，得稍微介绍一下其他历法。

1. 儒略历

儒略历(Julian Calendar)是现今公历的前身，用来取代罗马历(Roman Calendar)，于公元前 46 年被 Julius Caesar 采纳，公元前 45 年实现，约于公元 4 年至 1582 年之间广为各地采用。儒略历修正了罗马历隔三年设置一闰年的错误，改采四年一闰。

2. 格里高利历

格里高利历(Gregorian Calendar)改革了儒略历，由教宗 Pope Gregory XIII 于 1582 年颁行，将儒略历 1582 年 10 月 4 日星期四的隔天，订为格里高利历 1582 年 10 月 15 日星期五。

不过各个国家改历的时间并不相同，像英国、大英帝国(包含现今美国东部)改历的时间是在 1752 年 9 月初，因此在 Unix/Linux 中查询 1752 年月历，会发现 9 月平白少了 11 天，如图 13.1 所示。

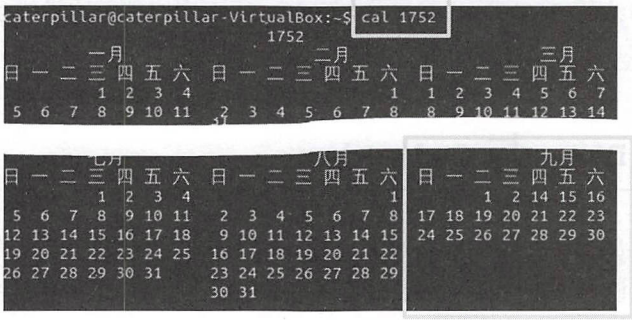


图 13.1　在 Linux 中查询 1752 年月历

3. ISO 8601 标准

在一些相对来说较新的时间日期 API 应用场合中，你可能会看过 ISO 8601。严格来说 ISO 8601 并非年历系统，而是时间日期表示方法的标准，用以统一时间日期的数据交换格式，例如 yyyy-mm-ddTHH:MM:SS.SSS、yyyy-dddTHH:MM:SS.SSS、yyyy-Www-dTHH:MM: SS.SSS 之类的标准格式。

ISO 8601 在数据定义上大部分与格里高利历相同，因而有些处理时间日期数据的程序或 API，为了符合时间日期数据交换格式的标准，会采用 ISO 8601。不过还是有些轻微差别。如在 ISO 8601 的定义中，19 世纪是指 1900 年至 1999 年(包含该年)，而格里高利历的 19 世纪是指 1801 年至 1900 年(包含该年)。

13.1.3　认识时区

在各种时间日期的议题中，时区(Time Zones)也许是最复杂的，每个地区的标准时间各不相同，因为这涉及地理、法律、经济、社会甚至政治等问题。

从地理上来说，由于地球是圆的，基本上一边白天另一边就是夜晚，为了让人们对时间的认知符合作息，因而设置了 UTC 偏移(Offset)。大致上，经度每 15 度是偏移一小时，考虑了 UTC 偏移的时间表示上，通常会标识 Z 符号。

不过有些国家的领土横跨经度大，一个国家有多个时间反而造成困扰，因而不采取每 15 度偏移一小时的作法，像美国有四个时区，而中国、印度只采单一时区。

除了时区考虑之外，有些高纬度国家，夏季、冬季日照时间差异很大，为了节省能源会尽量利用夏季日照，因而实施日光节约时间(Daylight Saving Time)，也称为夏令时(Summer Time)，就是在实施的第一天，让白天的时间增加一小时，而最后一天结束后再调整一小时回来。中国也曾实施过夏令时，后来因为没太大实质作用而取消，现在许多开发者多半不知道夏令时，偶尔会因此踩到误区。举例来说，如 1975 年 3 月 31 日 23 时 59 分 59 秒的下一秒，是从 1975 年 4 月 1 日 1 时 0 分 0 秒开始。

提示 »» 既然时区会涉及地理、法律、经济、社会甚至政治等问题，这也就表示随着时间的推进，不同时区的定义就得修正，例如某个国家或地区后来决定取消夏令时之类的，像 JDK 的时区信息，会随着不同版本的 JDK 发布而更新，你也可以通过 Timezone Updater Tool 来进行更新：http://www.oracle.com/technetwork/java/javase/downloads/tzupdater-download-513681.html

如果你得认真面对时间日期处理，认识以上的基本信息是必要的，至少你应该知道，一年

的毫秒数绝对不是单纯的 365×24×60×60×1000，更不应该基于这类错误的观念来进行时间与日期运算。

13.2 认识 Date 与 Calendar

虽然从 JDK8 开始提出了新日期与时间处理 API，然而现存系统中仍有不少程序代码，使用了 JDK8 出现前就已存在的 **java.util.Date** 与 **java.util.Calendar** 等 API，因此，认识它们才能了解如何使用 JDK8 新日期与时间处理 API。

13.2.1 时间轴上瞬间的 Date

如果想要取得系统时间，方法之一是使用 **System.currentTimeMillis()** 方法，返回的是 long 类型整数，代表 1970 年 1 月 1 日 0 时 0 分 0 秒 0 毫秒至今经过的毫秒数，也就是时间起点与前面谈到的 Unix 时间起点是相同的，以此方法取得的是机器的时间观点，代表着时间轴上的某一瞬间。然而这一长串 epoch 毫秒数不是人类的时间观点，对人类来说没有阅读上的意义。有人会使用 **Date** 实例来取得系统时间描述，不过 **Date** 也是偏向机器的时间观点。例如：

DateCalendar DateDemo.java

```
package cc.openhome;

import java.util.*;
import static java.lang.System.*;

public class DateDemo {
    public static void main(String[] args) {
        Date date1 = new Date(currentTimeMillis());
        Date date2 = new Date();

        out.println(date1.getTime());
        out.println(date2.getTime());
    }
}
```

Date 有两个构造函数可以使用，一个可使用 epoch 毫秒数构建，另一个为无自变量构造函数，内部亦是使用 System.currentTimeMillis() 取得 epoch 毫秒数，调用 **getTime()** 可取得内部保存的 epoch 毫秒数值。范例执行结果如下：

```
1398741380778
1398741380778
```

Date 类是从 JDK1.0 就已存在的 API，除了范例中使用的两个构造函数外，其他版本的构造函数都已废除。除此之外，getTime() 之外的 getXXX() 方法都废弃了，而 **setTime()**(用来设置 epoch 毫秒数)外的 setXXX() 方法也都废弃了，也就是说，**Date** 实例基本上建议只用来当作时间轴上的某一瞬间，也就是 1970 年 1 月 1 日 0 时 0 分 0 秒至今经过的毫秒数，其他对时间日期字段的设定与取得，建议通过稍后会介绍的 Calendar 来进行。

Date 的 toString() 虽然可以按内含的 epoch 毫秒数，计算并返回人类可懂的字符串时间格

式 dow mon dd hh:mm:ss zzz yyyy 分别是星期(dow)、月(mon)、日(dd)、时(hh)、分(mm)、秒(ss)、时区(zzz)与公元年(yyyy)，你没有方法改变这个格式，实际上因为 Date 实例的时区无法变换，也不建议使用 **toString()**来得知年月日等字段信息。toLocaleString()、toGMTString()这两个方法也被废弃了，也就是说，有关于字符串时间格式的处理，不再是 **Date** 的职责。

13.2.2 格式化时间日期的 DateFormat

有关字符串时间格式的处理，职责落到了 **java.text.DateFormat** 身上，DateFormat 是个抽象类，其操作类是 **java.text.SimpleDateFormat**，你可以直接构建 SimpleDateFormat 实例，或是使用 DateFormat 的 **getDateInstance()**、**getTimeInstance()**、**getDateTimeInstance()**等静态方法，用较简便方式按不同需求取得 SimpleDateFormat 实例。先来看看如何通过 DateFormat 的各种静态方法进行格式化：

DateCalendar DateFormatDemo.java

```
package cc.openhome;

import java.util.*;
import static java.lang.System.out;
import static java.text.DateFormat.*;

public class DateFormatDemo {
    public static void main(String[] args) {
        Date date = new Date();
        dateInstanceDemo(date);
        timeInstanceDemo(date);
        dateTimeInstanceDemo(date);
    }

    static void dateInstanceDemo(Date date) {
        out.println("getDateInstance() demo");
        out.printf("\tLONG: %s%n", getDateInstance(LONG).format(date));
        out.printf("\tSHORT: %s%n", getDateInstance(SHORT).format(date));
    }

    static void timeInstanceDemo(Date date) {
        out.println("getTimeInstance() demo");
        out.printf("\tLONG: %s%n", getTimeInstance(LONG).format(date));
        out.printf("\tMEDIUM: %s%n", getTimeInstance(MEDIUM).format(date));
        out.printf("\tSHORT: %s%n",getTimeInstance(SHORT).format(date));
    }

    static void dateTimeInstanceDemo(Date date) {
        out.println("getDateTimeInstance() demo");
        out.printf("\tLONG: %s%n",
           getDateTimeInstance(LONG, LONG).format(date));
        out.printf("\tMEDIUM: %s%n",
           getDateTimeInstance(SHORT, MEDIUM).format(date));
        out.printf("\tSHORT: %s%n",
           getDateTimeInstance(SHORT, SHORT).format(date));
    }
}
```

getDateInstance()、getTimeInstance()、getDateTimeInstance()等静态方法主要是取得不

同详细程度的日期时间，它们都重载了多个版本。范例中使用参数较多的版本，取得 DateFormat 实例时，也可以指定 Locale 实例，这会将日期时间格式化为指定的语系显示方式，Locale 的使用会在第 15 章介绍。来看范例显示结果：

```
getDateInstance() demo
      SHORT: 2018 年 3 月 10 日
      SHORT: 2018/03/10
getTimeInstance() demo
      LONG: 下午 5:11:22 [CST]
      MEDIUM: 下午 5:11:22
      SHORT: 下午 5:11
getDateTimeInstance() demo
      LONG: 2018 年 3 月 10 日 下午 5:11:22 [CST]
      MEDIUM: 2018/03/10 下午 5:11:22
      SHORT: 2010/03/10 下午 5:11
```

提示 »» DateFormat 的 getDateInstance()、getTimeInstance()等静态方法，基本上是返回 SimpleDateFormat 实例。不过根据 API 文件的说明，在某些特别的国家或区域设定下，有可能返回的不是 SimpleDateFormat 实例。

直接构建 SimpleDateFormat 的好处是，可使用模式字符串自定义格式。例如：

DateCalendar SimpleDateFormatDemo.java

```
package cc.openhome;

import java.text.*;
import java.util.*;

public class CurrentTime {
    public static void main(String[] args) {
        DateFormat dateFormat = new SimpleDateFormat(
                args.length == 0 ? "EE-MM-dd-yyyy" : args[0]);
        System.out.println(dateFormat.format(new Date()));
    }
}
```

这个范例可以让用户指定格式，或者使用默认格式"EE-MM-dd-yyyy"，"EE"表示星期格式设定，"MM"表示月份格式设定，"dd"表示日期格式设定，而"yyyy"是公元格式设定，每个字符设定都有其意义，可参考 SimpleDateFormat 的 API 说明，了解每个字符的设定意义。

SimpleDateFormat 还有个 **parse()** 方法，可以按构建 SimpleDateFormat 时指定的格式，将指定的字符串剖析为 Date 实例。例如：

DateCalendar HowOld.java

```
package cc.openhome;

import java.util.*;
import java.text.*;

public class HowOld {
    public static void main(String[] args) throws Exception {
        System.out.print("输入出生年月日(yyyy-mm-dd): ");
        DateFormat dateFormat = new SimpleDateFormat("yyyy-mm-dd");
```

```
            Date birthDate = dateFormat.parse(new Scanner(System.in).nextLine());
            Date currentDate = new Date();
            long life = currentDate.getTime() - birthDate.getTime();
            System.out.println("你今年的岁数为：" +
                    (life / (365 * 24 * 60 * 60 * 1000L)));
        }
}
```

这个程序可以让用户以 "yyyy-mm-dd" 的格式输入出生年月日，使用 DateFormat 的 parse() 方法将输入的字符串剖析为代表生日的 Date 后，再与代表现在时间的 Date 做运算，方法是调用 getTime()后相减，就是至今存活的毫秒数，与一年的毫秒数相除，看起来像是算出使用者的岁数了。执行结果如下：

```
输入出生年月日(yyyy-mm-dd)：1975-10-17
你今年的岁数为：42
```

不过！如 13.1.3 节结束前说到的，一年的毫秒数并不是如这个范例中，可以单纯地使用 365×24×60×60×1000 计算出来，这个范例只是简单示范，不应当这样计算用户岁数，实际上算出来的岁数也是错的，撰写这段的时间点是 2017/10/10，因为还没过 10/17，所以应当是 41 岁而已。

提示 »» 相对于 DateFormat 可进行日期时间格式化，java.text.NumberFormat 可用来进行数字格式化，它们都是 java.text.Format 的子类，两者使用上类似，第 15 章会谈到如何处理数字，以及 NumberFormat 的使用方式。

13.2.3 处理时间日期的 Calendar

Date 现在建议作为时间轴上的瞬时代表，要格式化时间日期则通过 DateFormat。如果想要取得某个时间日期信息，或者是对时间日期进行操作，可以使用 **Calendar** 实例。

Calendar 是个抽象类，**java.util.GregorianCalendar** 是其子类，操作了儒略历与格里高利历的混合历，通过 Calendar 的 **getInstance()**取得的 Calendar 实例，默认就是取得 GregorianCalendar 实例。例如：

```
jshell> Calendar calendar = Calendar.getInstance();
calendar ==> java.util.GregorianCalendar[time=1507626869508,ar ... SET=28800000,
DST_OFFSET=0]
```

取得 Calendar 实例后，可以使用 **getTime()**取得 Date 实例，如果想要取得年月日等日期时间字段，可以使用 **get()**方法并指定 Calendar 上的字段枚举常数。例如，想取得年、月、日字段，可用下面的程序：

```
jshell> calendar.get(Calendar.YEAR);
$2 ==> 2017

jshell> calendar.get(Calendar.MONTH);
$3 ==> 9

jshell> calendar.get(Calendar.DATE);
$4 ==> 10
```

撰写这个范例的时间是 2017/10/10，然而 calendar.get(Calendar.MONTH)取得的数字是 9，事实上这个数字是对应至 Calendar 在月份上的列举值，而列举值的一月是从 0 数字开始：

```
public final static int JANUARY = 0;
public final static int FEBRUARY = 1;
public final static int MARCH = 2;
public final static int APRIL = 3;
public final static int MAY = 4;
public final static int JUNE = 5;
public final static int JULY = 6;
public final static int AUGUST = 7;
public final static int SEPTEMBER = 8;
public final static int OCTOBER = 9;
public final static int NOVEMBER = 10;
public final static int DECEMBER = 11;
```

如果你要设定时间日期等字段，不要对 Date 设定，应该使用 Calendar，同样地，月份的部分请使用枚举常数设定。例如：

```
jshell> Calendar calendar = Calendar.getInstance();
calendar ==> java.util.GregorianCalendar[time=1507627008715,ar ... SET=28800000,DST_
OFFSET=0]

jshell> calendar.set(2017, Calendar.OCTOBER, 10);

jshell> calendar.get(Calendar.YEAR);
$3 ==> 2017

jshell> calendar.get(Calendar.MONTH);
$4 ==> 9

jshell> calendar.get(Calendar.DATE);
$5 ==> 10
```

在取得一个 Calendar 的实例后，可以使用 **add()**方法来改变 Calendar 的时间。例如：

```
calendar.add(Calendar.MONTH, 1);          // Calendar 的时间加 1 个月
calendar.add(Calendar.HOUR, 3);           // Calendar 的时间加 3 小时
calendar.add(Calendar.YEAR, -2);          // Calendar 的时间减 2 年
calendar.add(Calendar.DATE, 3);           // Calendar 的时间加 3 天
```

如果 calendar 设定的时间是 2017/2/28(非闰年)，calendar.add(Calendar.DATE, 1)的结果会是 2017/3/1，也就是相当于下一天应有的日期。如果打算只针对日期中某个字段加减，则可以使用 **roll()**方法，例如：

```
calendar.roll(Calendar.DATE, 1);          // 只对日字段加 1
```

如果 calendar 设定的时间是 2017/2/28，calendar.roll(Calendar.DATE, 1)的结果会是 2017/2/1，也就是只处理日字段的部分，因为当年 2 月没有 29 日，因而实际上就是回到 1 日。

Calendar 是个抽象类，GregorianCalendar 是其子类，操作了儒略历与格里高利历的混合历，因而可同时支持儒略历与格里高利历，默认的日历时间为格里高利历 1582 年 10 月 15 日星期五，因此如果运行以下的运算：

```
Calendar calendar = Calendar.getInstance();
calendar.set(1582, Calendar.OCTOBER, 15);
out.println(calendar.getTime());          // 显示格里高利历 1582 年 10 月 15 日
```

```
calendar.add(Calendar.DAY_OF_MONTH, -1);  // 往前一天
out.println(calendar.getTime());          // 显示儒略历 1582 年 10 月 4 日
```

使用 GregorianCalendar 对格里高利历 1582 年 10 月 15 日作减去一天的运算，实际上会来到儒略历 1582 年 10 月 4 日。日历时间可以使用 GregorianCalendar 的 **setGregorianChange()** 方法来修改，设为 Date(Long.MAX_VALUE) 就是纯粹的儒略历，设为 Date(Long.MIN_VALUE) 就是纯粹的格里高利历。

提示 »» 刚刚还是用了 Date 实例的 toString() 方法来显示时间信息(out.println() 会调用自变量的 toString())，这只是为了简化范例，观察到的结果也是正确的信息，不过，还是不建议用 Date 实例的 toString() 方法。

想要比较两个 Calendar 的时间日期先后，可以使用 **after()** 或 **before()** 方法。这里先回顾一下刚刚看到的 HowOld 范例，当时谈到，单纯地使用 365×24×60×60×1000 当作一年的毫秒数并用以计算用户的岁数是不对的，而应该使用 Calendar 的相关操作。例如：

DateCalendar CalendarUtil.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.Calendar;

public class CalendarUtil {
    public static void main(String[] args) {
        Calendar birth = Calendar.getInstance();
        birth.set(1975, Calendar.MAY, 26);
        Calendar now = Calendar.getInstance();
        out.printf("岁数: %d%n", yearsBetween(birth, now));
        out.printf("天数: %d%n", daysBetween(birth, now));
    }

    public static long yearsBetween(Calendar begin, Calendar end) {
        Calendar calendar = (Calendar) begin.clone();
        long years = 0;
        while (calendar.before(end)) {
            calendar.add(Calendar.YEAR, 1);
            years++;
        }
        return years - 1;
    }

    public static long daysBetween(Calendar begin, Calendar end) {
        Calendar calendar = (Calendar) begin.clone();
        long days = 0;
        while (calendar.before(end)) {
            calendar.add(Calendar.DATE, 1);
            days++;
        }
        return days - 1;
    }
}
```

注意，如果要在 Calendar 实例上进行 add() 之类的操作，记得这操作会修改 Calendar 实例本身，为了避免调用 yearsBetween()、daysBetween() 之后传入的 Calendar 自变量被修改，两个

方法中都对第一个自变量进行了 clone() 复制对象的动作。执行结果如下：

```
岁数：42
天数：15478
```

13.2.4 设定 TimeZone

前面在使用 Calendar 时，并没有使用时区信息，而是使用默认时区，可以使用 **java.util.TimeZone** 的 **getDefault()** 来取得默认时区信息。例如：

DateCalendar TimeZoneDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.TimeZone;

public class TimeZoneDemo {
    public static void main(String[] args) {
        TimeZone timeZone = TimeZone.getDefault();
        out.println(timeZone.getDisplayName());
        out.println("\t 时区 ID: " + timeZone.getID());
        out.println("\t 夏令时数: " + timeZone.getDSTSavings());
        out.println("\tUTC 偏移毫秒数: " + timeZone.getRawOffset());
    }
}
```

执行结果如下：

```
中国标准时间
   时区 ID: Asia/Shanghai
   夏令时数: 0
   UTC 偏移毫秒数: 28800000
```

如果想要取得指定时区的 TimeZone 实例，可以使用 ID 字符串，例如：

```
TimeZone taipeiTz = TimeZone.getTimeZone("Asia/Taipei");
TimeZone copenhagenTz = TimeZone.getTimeZone("Europe/Copenhagen");
```

可用的 ID 可以使用 TimeZone.getAvailableIDs() 来取得，它会返回 String[]。Calendar 在调用 getInstance() 时，可以指定 TimeZone，如果已经取得 Calendar 实例，也可以通过 setTimeZone() 方法设定 TimeZone。例如，想知道现在哥本哈根的时间，可以如下撰写程序：

DateCalendar TimeZoneDemo2.java

```
package cc.openhome;

import java.util.*;
import static java.lang.System.out;

public class TimeZoneDemo2 {
    public static void main(String[] args) {
        TimeZone taipeiTz = TimeZone.getTimeZone("Asia/Taipei");
        Calendar calendar = Calendar.getInstance(taipeiTz);
        showTime(calendar);

        TimeZone copenhagenTz = TimeZone.getTimeZone("Europe/Copenhagen");
```

```
        calendar.setTimeZone(copenhagenTz);
        showTime(calendar);
    }

    static void showTime(Calendar calendar) {
        out.print(calendar.getTimeZone().getDisplayName());;
        out.printf(" %d:%d%n",
                calendar.get(Calendar.HOUR),
                calendar.get(Calendar.MINUTE));
    }
}
```

执行结果如下：

```
中国标准时间 0:19
中欧时间 6:19
```

13.3 新时间日期 API

在 13.2 节中已经介绍，Date 实例真正代表的并不是日期，最接近的概念应该是时间轴上特定的一瞬间，时间精度是毫秒，也就是 UTC 时间 1970 年 1 月 1 日 0 时 0 分 0 毫秒至某个特定瞬时的毫秒差。Date 的 setTime()方法没有被废弃，也就是说，Date 状态仍是可变的，如果在 API 之间传递它而不想改变它的状态，就得祈祷 API 别去动它。

Calendar 提供了一些计算日期时间的方法，不过，使用 Calendar 太麻烦了，得用一堆枚举常数，例如 YEAR、MONTH、DAY_OF_MONTH、HOUR 等，不然就得小心那些从 0 开始计算的日期时间，如月份。此外，Calendar 状态可变，有时也会造成问题。

JDK8 中有了新的时间日期处理 API，规格书为 JSR310，在处理上可以更加简便。

13.3.1 机器时间观点的 API

Date 名称看来像人类的时间概念，实际却是机器的时间概念，混淆机器与人类时间观点会引发的问题之一如夏令时。例如 1975 年 3 月 31 日 23 时 59 分 59 秒的下一秒，是从 1975 年 4 月 1 日 1 时 0 分 0 秒开始，如果你试着如下撰写程序：

```
shell> Calendar calendar = Calendar.getInstance();
calendar ==> java.util.GregorianCalendar[time=1507627530171,ar ...
SET=28800000,DST_OFFSET=0]

jshell> calendar.set(1975, Calendar.MARCH, 31, 23, 59, 59);

jshell> calendar.getTime();
$3 ==> Mon Mar 31 23:59:59 CST 1975

jshell> calendar.add(Calendar.SECOND, 1);     // 增加一秒

jshell> calendar.getTime();
$5 ==> Tue Apr 01 01:00:00 CDT 1975
```

Calendar 的 getTime()返回 Date 实例，如果系统设置为夏令时的时区，toString()返回的

字符串描述会是"Tue Apr 01 **01:00:00** CDT 1975"，而不是"Tue Apr 01 **00:00:00** CDT 1975"。

由于现在中国已经不实施夏令时了，许多开发者并不知道过去有过夏令时，在取得 Date 实例后，被名称 **Date** 误导它们代表日期，却看到显示为"Tue Apr 01 01:00:00 CDT 1975"时，就会感到困惑。就如之前讲过的，你不该使用 **Date** 实例的 **toString()** 来得知人类观点的时间信息，而且 **Date** 实例应该只代表机器观点的时间信息，真正可靠的信息只有内含的 epoch 毫秒数。如果取得 **Date** 实例，下一步该获取时间信息应该是通过 **Date** 的 **getTime()** 取得 epoch 毫秒数，这样就不会混淆。以下范例可以看出，165513599154 下一秒就是 **165513600154**：

```
jshell> Calendar calendar = Calendar.getInstance();
calendar ==> java.util.GregorianCalendar[time=1507627634154,ar ...
SET=28800000,DST_OFFSET=0]

jshell> calendar.set(1975, Calendar.MARCH, 31, 23, 59, 59);

jshell> calendar.getTime().getTime();
$3 ==> 165513599154

jshell> calendar.add(Calendar.SECOND, 1);     // 增加一秒

jshell> calendar.getTime().getTime();
$5 ==> 165513600154
```

JDK8 新时间日期处理 API 中最重要的，就是清楚地将机器对时间的概念与人类对时间的概念区分开来了，让机器与人类对时间概念的界线变得分明。新时间日期处理 API 的主要套件命名为 **java.time**。对于机器相关的时间概念，设计了 **Instant** 类，用以代表自定义的 Java epoch(1970 年 1 月 1 日)之后的某个时间点历经的毫秒数，精确度基本上是毫秒，但可添加奈秒(nanosecond)精度的修正数值。

提示 »» 为了避免时间定义上的模糊，JSR310 定义了自己的时间度量(Time-scale)，例如 Java epoch、年历上的一天是 86400 秒等，可以在 Instant 的 API 文件查询得知其如何定义时间的度量方式。

使用 Instant 的静态方法 **now()** 可以取得代表 Java epoch 毫秒数的 Instant 实例，**ofEpochMilli()** 可以指定 Java epoch 毫秒数，**ofEpochSecond()** 则可以指定秒数。在取得 Instant 实例后，可以使用 **plusSeconds()**、**plusMillis()**、**plusNanos()**、**minusSeconds()**、**minusMillis()**、**minusNanos()** 来做时间轴上的运算，Instant 实例本身不会变动，这些操作都会返回新的 Instant 实例，代表运算后的瞬时。

在新旧 API 兼容上，如果你取得了 Date 实例，而你想要改用 Instant，可以调用 Date 实例的 **toInstant()** 方法来取得；如果你有个 Instant 实例，则可以使用 Date 的静态方法 **from()** 转为 Date。

提示 »» 如果访客在你的留言版留下信息，你该怎么记录信息建立的时间呢？要用机器的时间观点？还是人类的时间观点？可以参考一下 *Java 8 LocalDateTime vs Instant* 文章(稍后就会介绍 LocalDateTime)：
http://ingramchen.io/blog/2014/04/java-8-local-date-time-vs-instant.html

13.3.2 人类时间观点的 API

人类在时间的表达上有时只需要日期，有时只需要时间，有时会同时表达日期与时间，而且通常不会特别声明时区，也很少在意夏令时，可能只会提及年、月、年月、月日等。简而言之，人类在时间概念的表达大多是笼统、片段的信息。

1. LocalDateTime、LocalDate 和 LocalTime

对于片段的日期时间，JDK8 新时间与日期 API 有 **LocalDateTime**(包括日期与时间)、**LocalDate**(只有日期)、**LocalTime**(只有时间)等类来定义，这些类基于 ISO 8601 年历系统，是不具时区的时间与日期定义。

LocalDateTime、LocalDate、LocalTime 等类名称开头为 Local，表示它们都只是对时间的描述，并没有时区信息。然而，对于 LocalDate，如果设定了不存在的日期，例如 LocalDate.of(2018, 2, 29)会抛出 **DateTimeException**，因为 2018 年并非闰年。不过，对于 LocalDateTime.of(2018, 4, 1, 0, 0, 0)，由于没有时区信息，程序无从判断这个时间是否不存在，就不会抛出 DateTimeException。

提示»» 虽然就 API 本身来说，LocalDate、LocalTime、LocalDateTime 本身不带时区信息，不过程序运行的时区就暗示着是 LocalDate、LocalTime、LocalDateTime 的时区，因为人们若不特别提及时区，其实就是指本地时区居多。

2. ZonedDateTime 和 OffsetDateTime

如果时间日期需要带有时区，可以基于 LocalDateTime、LocalDate、LocalTime 等来补齐缺少的信息：

DateCalendar ZonedDateTimeDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.time.*;

public class ZonedDateTimeDemo {
    public static void main(String[] args) {
        LocalTime localTime = LocalTime.of(0, 0, 0);
        LocalDate localDate = LocalDate.of(1975, 4, 1);
        ZonedDateTime zonedDateTime = ZonedDateTime.of(
                localDate, localTime, ZoneId.of("Asia/Taipei"));

        out.println(zonedDateTime);
        out.println(zonedDateTime.toEpochSecond());
        out.println(zonedDateTime.toInstant().toEpochMilli());
    }
}
```

可以从执行结果看到，当补上时区信息后，如果组合起来的时间实际上不存在，**ZonedDateTime** 会自动更正，并不会抛出异常：

```
1975-04-01T01:00+09:00[Asia/Taipei]
165513600
165513600000
```

提示>>> 用户自行设定的时间，可以使用 LocalDate、LocalTime、LocalDateTime 来表示。为了避免用户设定了不存在的时间，建议操作接口上，可以使用日期时间选择器(Date Time Picker)组件，只显示存在的时间让用户选取。

在新的时间与日期 API 中，UTC 偏移量与时区的概念是分开的。OffsetDateTime 单纯代表 UTC 偏移量，使用 ISO 8601，如果有 LocalDateTime、LocalDate、LocalTime，也可以在分别补齐必要信息后，取得 UTC 偏移量：

```
LocalDate nowDate = LocalDate.now();
LocalTime nowTime = LocalTime.now();
OffsetDateTime offsetDateTime =
        OffsetDateTime.of(nowDate, nowTime, ZoneOffset.UTC);
```

ZonedDateTime 与 OffsetDateTime 间可以通过 toXXX()方法互转，Instant 通过 **atZone()**与 **atOffset()**转为 ZonedDateTime 与 OffsetDateTime，ZonedDateTime 与 OffsetDateTime 也都可以通过 **toInstant()**取得 Instant，ZonedDateTime 与 OffsetDateTime 都有 **toLocalDate()**、**toLocalTime()**、**toLocalDateTime()**方法可以取得 LocalDate、LocalTime 与 LocalDateTime。

3. Year、YearMonth、Month 和 MonthDay

如果只想表示 2018 年，可以使用 Year；如果想表示 2018/5，可以使用 YearMonth；如果只想表示 5 月，可以使用 Month；如果想表示 5/4，可以使用 MonthDay。其中，Month 是 enum 类型。要注意的是，如果你想要取得代表月份的数字，不要使用 **ordinal()**方法，因为 ordinal()是 enum 在定义时的顺序，从 0 开始，想要取得代表月份的数要通过 **getValue()**方法。

DateCalendar MonthDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.time.Month;

public class MonthDemo {
    public static void main(String[] args) {
        for(Month month : Month.values()) {
            out.printf("original: %d\tvalue: %d\t%s%n",
                    month.ordinal(), month.getValue(), month);
        }
    }
}
```

执行结果如下：

```
original: 0         value: 1  JANUARY
original: 1         value: 2  FEBRUARY
original: 2         value: 3  MARCH
original: 3         value: 4  APRIL
original: 4         value: 5  MAY
original: 5         value: 6  JUNE
original: 6         value: 7  JULY
original: 7         value: 8  AUGUST
original: 8         value: 9  SEPTEMBER
```

```
original: 9         value: 10 OCTOBER
original: 10        value: 11 NOVEMBER
original: 11        value: 12 DECEMBER
```

提示>>> 在“人类时间观点的API”这节介绍到的类，都有个 now()方法，如果不指定任何自变量，会使用默认的 Clock 对象，它会使用默认时区的系统时钟来取得目前时间。实际上可以在必要时，指定给 now()方法一个自定义的 Clock 对象，这样 now()就会根据你给的时钟来产生目前的时间，详情可参考 Clock 的 API 文件。

13.3.3 对时间的运算

如果要知道某个日期起加上 5 天、6 个月、3 周后的日期时间会是什么，并使用指定的格式输出。使用 Calendar 的话，会需要如下的计算：

```
Calendar calendar = Calendar.getInstance();
calendar.set(1975, Calendar.MAY, 26, 0, 0, 0);
calendar.add(Calendar.DAY_OF_MONTH, 5);
calendar.add(Calendar.MONTH, 6);
calendar.add(Calendar.WEEK_OF_MONTH, 3);
SimpleDateFormat df = new SimpleDateFormat("E MM/dd/yyyy");
out.println(df.format(calendar.getTime()));
```

1. TemporalAmount

JDK8 新日期时间处理实现了流畅 API(Fluent API)的概念，写的程序会轻松且流畅易读：

```
out.println(
    LocalDate.of(1975, 5, 26)
             .plusDays(5)
             .plusMonths(6)
             .plusWeeks(3)
             .format(ofPattern("E MM/dd/yyyy"))
);
```

提示>>> 有关流畅 API 的概念，可以参考《写一手流畅的 API》：
http://openhome.cc/Gossip/Programmer/FluentAPI.html

其中的 ofPattern()其实是 **java.time.format.DateTimeFormatter** 的静态方法，可以查看 API 文件了解格式化的方式。LocalDate 的 plusDays()、plusMonths()、plusWeeks()只是时间运算时一些常用的指定方法。当然，时间运算的需求很多，不可能列出全部的 plusXXX()方法。对于时间计量，新时间与日期 API 以类 **Duration** 来定义，可用于计量天、时、分、秒的时间差，精度调整可以达到纳秒等级，而秒的最大值可以是 long 类型可保存值。对于年、月、星期、日的日期差，则使用 **Period** 类定义。上例可以改为：

```
out.println(
    LocalDate.of(1975, 5, 26)
             .plus(ofDays(5))
             .plus(ofMonths(6))
             .plus(ofWeeks(3))
             .format(ofPattern("E MM/dd/yyyy"))
);
```

其中 ofDays()、ofMonths()、ofWeeks()其实是 Period 的静态方法，它们会返回 Period 实

例。实际上，**plus()**方法接受 **java.time.temporal.TemporalAmount** 实例，而 TemporalAmount 的操作类也就是 Period 与 Duration，因此，实际上 plus()方法也可以接受 Duration 实例来计算。

前面看过的 HowOld 范例，也可以使用新时间与日期 API 改写如下：

DateCalendar HowOld2.java

```
package cc.openhome;

import java.time.*;
import java.util.Scanner;
import static java.lang.System.out;

public class HowOld2 {
    public static void main(String[] args) {
        out.print("输入出生年月日(yyyy-mm-dd): ");
        LocalDate birth = LocalDate.parse(new Scanner(System.in).nextLine());
        LocalDate now = LocalDate.now();
        Period period = Period.between(birth, now);
        out.printf("你活了 %d 年 %d 月 %d 日%n",
                period.getYears(), period.getMonths(), period.getDays());
    }
}
```

这次不只计算岁数，也计算了月数与日数，即使如此，整个程序仍非常简洁，执行的范例之一如下：

```
输入出生年月日(yyyy-mm-dd): 1975-10-17
你活了 41 年 11 月 23 日
```

提示»» Period 与 Duration 乍看有些难区别，简单来说，Period 是日期差，between()方法只接受 LocalDate，不表示比“日”更小的单位。然而 Duration 是时间差，between()可以接受 Temporal 操作对象(马上就会介绍)，也就是说可以用 LocalDate、LocalTime、LocalDateTime 来计算 Duration，不表示比“天”更大的单位。

2. TemporalUnit

plus()方法另一重载版本，接受 **java.time.temporal.TemporalUnit** 实例，**java.time.temporal.ChronoUnit** 是 TemporalUnit 实作类，使用 enum 实作。因此，上例也可以使用以下方式实作，就阅读上会更符合人类的习惯。

```
out.println(
    LocalDate.of(1975, 5, 26)
             .plus(5, DAYS)
             .plus(6, MONTHS)
             .plus(3, WEEKS)
             .format(ofPattern("E MM/dd/yyyy"))
);
```

TemporalUnit 定义了 between()等方法，例如，使用操作类 ChronoUnit 的枚举实例来操作之前的 CalendarUtil 范例，就非常方便：

```
LocalDate birth = LocalDate.of(1975, 5, 26);
LocalDate now = LocalDate.now();
out.printf("岁数: %d%n", ChronoUnit.YEARS.between(birth, now));
out.printf("天数: %d%n", ChronoUnit.DAYS.between(birth, now));
```

3. Temporal

你应该可以察觉到，JDK8 新日期时间 API，将时间的运算行为抽取出来独立定义，放置在 **java.time.temporal** 套件之中，这是基于 API 实作弹性上的考虑，与人类时间或机器时间概念无关。Instant、LocalDate、LocalDateTime、LocalTime、OffsetDateTime、ZonedDateTime 等类，都操作了 **Temporal** 接口，如图 13.2 所示。

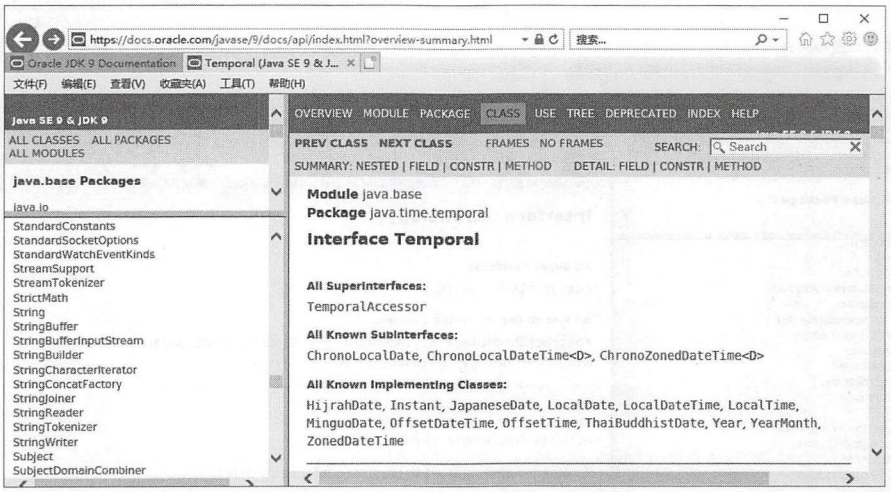


图 13.2 Temporal 接口与操作类

实际上，刚才看到的 plus() 方法就是定义在 Temporal 接口上，相对于 plus() 也有两个重载版本的 minus() 方法：

- plus(TemporalAmount amount)
- plus(long amountToAdd, TemporalUnit unit)
- minus(TemporalAmount amount)
- minus(long amountToSubtract, TemporalUnit unit)

提示»» 如果需要更复杂的调整，可以使用 with(TemporalAdjuster adjuster)，细节可参考 emporalAdjuster 的 API 文件。

4. TemporalAccessor

TemporalAccessor 定义了只读的时间对象(像是日期、时间、偏移量等)读取操作，实际上 Temporal 是 TemporalAccessor 子接口，增加了对时间的处理操作，像是 plus()、minus()、with() 等方法。有趣的是，之前看过的 MonthDay 是只读的，也就是仅操作了 TemporalAccessor 接口，为什么呢？在 MonthDay 的 API 文件有说明，因为有闰年问题，在缺少“年”的信息下，如果 MonthDay 可进行 plus() 操作，那么 2 月 28 日加一天会是 2 月 29 日或是 3 月 1 日就无法定义了。

13.3.4 年历系统设计

JDK8 采单一年历系统设计，也就是说，**java.time** 套件中的类在需要实行年历系统时都是采用单一的 ISO8601 年历系统；那么，如果需要其他年历系统时呢？需要明确实行 java.time.chrono 中等操作了 **java.time.chrono.Chronology** 接口的类，如图 13.3 所示。

其中，**MinguoChronology** 与之搭配的主要类是 **MinguoDate**，操作了 Temporal、

TemporalAdjuster 与 **java.time.chrono.ChronoLocalDate** 接口，前面介绍过的 LocalDate 类也操作了 ChronoLocalDate 接口，下面来看个简单的范例。

```
LocalDate birth = LocalDate.of(1975, 5, 26);
MinguoDate mingoBirth = MinguoDate.from(birth);
out.println(mingoBirth);
```

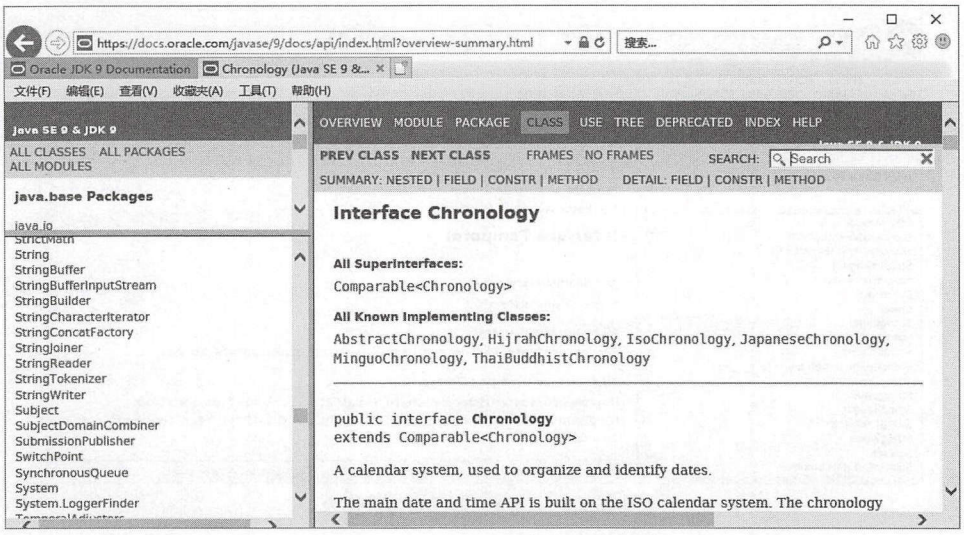


图 13.3 Chronology 接口与操作类

如果想要同时表示 **MinguoChronology** 的日期与时间，可以如下取得 **ChronoLocalDateTime<MinguoDate>**：

```
out.println(
    MinguoDate.of(64, 5, 1)
              .atTime(LocalTime.of(3, 30, 0)));
```

实际上，前面介绍过的 LocalDateTime，也操作了 ChronoLocalDateTime 接口，想要了解如何自定义年历系统，从 MinguoChronology 的原始码中研究，是个不错的起点。

提示»» Java 官方教学文件 *Java Tutorial* 中的 *Standard Calendar* 也对 Java 时间与日期处理做了不错的介绍，可以参考看看：http://docs.oracle.com/javase/tutorial/datetime/iso/

13.4 重点复习

GMT 时间常不严谨(且有争议性)地当成是 UTC。现在 GMT 已不作为标准时间使用。在 1972 年导入 UTC 之前，GMT 与 UT 是相同的。1967 年定义的国际原子时，将秒的国际单位定义为铯原子辐射振动 9192631770 周耗费的时间，时间从 UT 的 1958 年开始同步。

由于基于铯原子振动定义的秒长是固定的，然而地球自转会越来越慢，这会使得实际上 TAI 时间，会不断超前基于地球自转的 UT 系列时间，为了保持 TAI 与 UT 时间不要差距过大，因而提出了具有折中修正版本的世界协调时间，常简称为 UTC。UTC 经过了几次的时间修正，

非卖品，仅供非商业用途或交流学习使用

为了简化日后对时间的修正，1972 年 UTC 采用了闰秒修正。

Unix 系统的时间表示法，定义为 UTC 时间 1970 年(Unix 元年)1 月 1 日 00:00:00 为起点而经过的秒数，不考虑闰秒修正，用以表达时间轴上某一瞬间。

Epoch 为某个特定时代的开始，时间轴上某一瞬间。

格里高利历改革了儒略历，将儒略历 1582 年 10 月 4 日星期四的隔天，订为格里高利历 1582 年 10 月 15 日星期五。不过各个国家改历的时间并不相同，像英国、大英帝国(包含现今美国东部)改历的时间是在 1752 年 9 月初，因此在 Unix/Linux 中查询 1752 年月历，会发现 9 月平白少了 11 天。

严格来说 ISO 8601 并非年历系统，而是时间日期表示方法的标准，用以统一时间日期的数据交换格式，ISO 8601 在数据定义上大部分与格里高利历相同，不过还是有些轻微差别。如在 ISO 8601 的定义中，19 世纪是指 1900 年至 1999 年(包含该年)，而格里高利历的 19 世纪是指 1801 年至 1900 年(包含该年)。

为了让人们对时间的认知符合作息，因而设置了 UTC 偏移，大致上来说，经度每 15 度是偏移一小时。不过有些国家的领土横跨的经度很大，一个国家有多个时间反而造成困扰，因而不采取每 15 度偏移一小时的做法，像美国有四个时区，而中国、印度采用单一时区。

有些高纬度国家，夏季、冬季日照时间差异很大，为了节省能源会尽量利用夏季日照，因而实施日光节约时间，也称为夏令时。

如果你得认真面对时间日期处理，认识以上的基本信息是必要的，至少你应该知道，一年的毫秒数绝对不是单纯的 365×24×60×60×1000，更不应该基于这类错误的观念来进行时间与日期运算。

如果想要取得系统时间，方法之一是使用 `System.currentTimeMillis()` 方法，返回的是 `long` 类型整数，代表 1970 年 1 月 1 日 0 时 0 分 0 秒 0 毫秒至今经过的毫秒数，也就是时间起点与前面谈到的 Unix 时间起点是相同的，以此方法取得的是机器的时间观点，代表着时间轴上的某一瞬间。然而这一长串 epoch 毫秒数不是人类的时间观点，对人类来说没有阅读上的意义。有人会使用 `Date` 实例来取得系统时间描述，不过 `Date` 也是偏向机器的时间观点，`Date` 实例基本上建议只用来当作时间轴上的某一瞬间。

JDK8 新时间日期处理 API 中最重要的，就是清楚地将机器对时间的概念与人类对时间的概念区隔开来，让机器与人类对时间概念的界线变得分明。

13.5 课后练习

1. 使用 `Calendar` 撰写程序，使结果显示如图 13.4 所示的日历。

2018-05-07 星期一

日	一	二	三	四	五	六
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

图 13.4 Calendar 撰写程序显示结果

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

2. 使用 JDK8 新时间日期 API 撰写程序，使结果显示如图 13.5 所示的日历。

2018年05月07日 星期一

日	一	二	三	四	五	六
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

图 13.5　JDK8 新时间日期 API 撰写程序显示结果

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

Chapter 14 NIO 与 NIO2

学习目标

- 认识 NIO
- 使用 Channel 与 Buffer
- 使用 NIO2 文件系统

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

14.1 认识 NIO

第 10 章介绍了基于 InputStream、OutputStream、Reader、Writer 的输入/输出，对于高级输入/输出处理，Java 从 JDK1.4 开始提供了 NIO(New IO)，而 Java SE 7 之后又提供了 NIO2，认识与善用这些高级输入/输出处理 API，对于提高输入/输出的处理效率会有很大的帮助。

14.1.1 NIO 概述

InputStream、OutputStream 的输入/输出，基本上是以字节为单位进行低层次处理，虽然你得直接面对字节数组，但实际上多半是对字节数组中整个区块进行处理。例如，在 10.1.1 节中看过的 dump()方法，实际上是整块数据读入后又整块数据写出，然而必须处理 byte[]、记录读取的字节数、指定写出的 byte[]起点与字节数：

```
public static void dump(InputStream src, OutputStream dest)throws IOException {
    try (src; dest) {
        byte[] data = new byte[1024];
        int length;
        while ((length = src.read(data)) != -1) {
            dest.write(data, 0, length);
        }
    }
}
```

虽然 java.io 套件中也有一些装饰(Decorator)类，例如 DataInputStream、DataOutputStream、BufferedReader 与 BufferedWriter 等，不过，若只要对字节或字符串中感兴趣的区块进行处理，这些类就不见得适合，必须自行撰写 API 或寻找相关的链接库来处理索引、标记等细节。

相对于串流输入/输出使用 InputStream、OutputStream 来衔接数据源与目的地，NIO 使用频道(Channel)来衔接数据节点。在处理数据时，NIO 可以设定缓冲区(Buffer)容量，在缓冲区中对感兴趣的数据区块进行标记，如标记读取位置、数据有效位置，对于这些区块标记，提供了 clear()、rewind()、flip()、compact()等高级操作。举例来说，上面的 dump()方法，若使用 NIO 的话，可以如下撰写：

```
public static void dump(ReadableByteChannel src, WritableByteChannel dest)
                    throws IOException {
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    try(src; dest) {
        while(src.read(buffer) != -1) {
            buffer.flip();
            dest.write(buffer);
            buffer.clear();
        }
    }
}
```

稍后会介绍 API 的细节。在这段程序示范中，你只要确认有将数据从 Channel 中读入 Buffer(read()方法不返回-1)，使用高级 filp()方法标记 Buffer 中读入数据的所在区块，然后 Buffer 中的数据写到另一个 Channel，最后使用 clear()方法清除 Buffer 中的标记。在这个过程中，用户不用接触 byte[]的相关细节。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示 »» NIO 实际上包括更多的观念，然而认识 Channel 与 Buffer 是使用 NIO 的起点，也是这一节的重点。完整的 NIO 功能说明，例如 Selector 的使用，可以参考 NIO 专书的介绍。

14.1.2 Channel 架构与操作

NIO 中 Channel 相关接口与类位于 **java.nio.channels** 套件之中，**Channel** 接口是 AutoClosable 的子接口，因此都可以使用 JDK7 之后的尝试关闭资源语法。Channel 接口上主要新增了 **isOpen()** 方法，用来确认 Channel 是否开启。对 NIO 初学者来说，可以先认识图 14.1 所示的 Channel 继承架构。

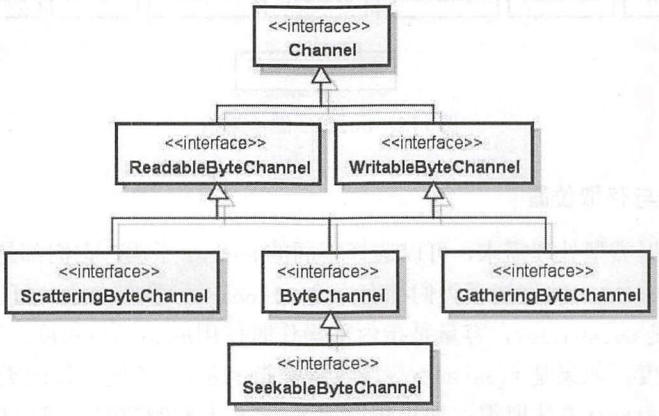


图 14.1 Channel 继承架构

ReadableByteChannel 定义了 **read()** 方法，负责将 ReadableByteChannel 中的数据读取至 ByteBuffer；**WritableByteChannel** 定义了 **write()** 方法，负责将 ByteBuffer 的数据写到 WritableByteChannel 中；ScatteringByteChannel 定义的 read() 方法，可以将 ScatteringByteChannel 分配到 ByteBuffer 数组中；GatheringByteChannel 定义的 write() 方法，可以将 ByteBuffer 数组的数据写到 GatheringByte-Channel。

ByteChannel 没有定义任何方法，单纯继承了 ReadableByteChannel 与 WritableByteChannel 的行为，ByteChannel 的子接口 SeekableByteChannel 可以读取与改变下一个要存取数据的位置。

提示 »» JDK7 在 java.nio.channels 中增加了一些 Channel 子接口与相关操作类，例如 Asynchronous Channel 接口、AsynchronousFileChannel 类等，可以查看 API 文件了解继承架构。

在 API 文件上可以看到 Channel 的操作类，不过都是抽象类，不能直接实例化，想要取得 Channel 的操作对象，可以使用 **Channels** 类，前面定义了静态方法 **newChannel()**，可以从 InputStream、OutputStream 分别建立 ReadableByteChannel、WritableByteChannel，有些 InputStream、OutputStream 实例本身也有方法可以取得 Channel 实例。举例来说，FileInputSteam、FileOutputStream 都有个 getChannel() 方法可以分别取得 FileChannel 实例(操作了 SeekableByteChannel、Gathering-ByteChannel、ScatteringByteChannel 接口)。

如果你已经有相关的 Channel 实例，也可以通过 Channels 上其他 newXXX() 静态方法，取得 InputStream、OutputStream、Reader、Writer 实例。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

14.1.3 Buffer 架构与操作

在 NIO 设计中，数据都是在 **java.nio.Buffer** 中处理，Buffer 是个抽象类，定义了 clear()、flip()、reset()、rewind()等对数据区块的高级操作，这类操作返回类型都是 Buffer，实际上返回 this。因此在需要连续高级操作时，可以形成管线操作风格，Buffer 的类继承架构如图 14.2 所示，它们都是抽象类。

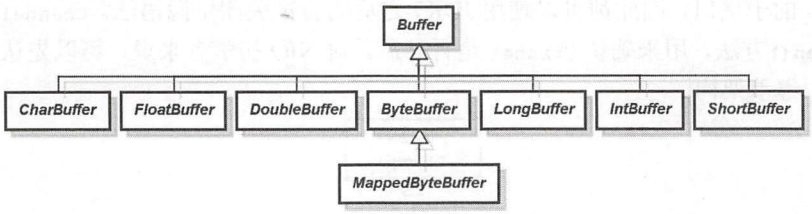


图 14.2 Buffer 继承架构

1. 容量、界限与存取位置

根据不同的数据类型处理需求，可以选择不同的 Buffer 子类，它们都是抽象类，因此不能直接实例化，然而 Buffer 的直接子类们都有一个 allocate()静态方法，可以指定 Buffer 容量(Capacity)。如果是 ByteBuffer，容量是指内部操作时使用的 byte[]长度；如果是 CharBuffer，容量是指 char[]长度；如果是 FloatBuffer，容量是指 float[]长度。依此类推，Buffer 的容量大小可以使用 **capacity()**方法取得；如果想取得 Buffer 内部的数组，可以使用 array()方法；如果你有个数组想要转为某个 Buffer 子类实例，每个 Buffer 子类实例都有 **wrap()**静态方法可以提供这项服务。

如果使用 ByteBuffer，它还有一个 **allocateDirect()**方法，相对于 allocate()方法配置的内存是由 JVM 管理，allocateDirect()会利用操作系统的原生 I/O 操作，试着避免 JVM 的中介转接，理论上会比 allocate()配置的内存更有效。不过 allocateDirect()在配置内存时会耗用较多系统资源，因此建议只用在大型、存活长的 ByteBuffer 对象，并能观察出明显功能差异的场合。想要知道 Buffer 是否为直接配置，可以通过 **isDirect()**得知。

Buffer 是个容器，填装的数据不能超过它的容量，实际可读取或写入的数据界限(Limit)索引值可以由 limit()方法得知或设定。举例来说，容量为 1024 字节的 ByteBuffer，ReadableByteChannel 对其写入了 512 字节，那么 limit()应该设为 512，至于下一个可读取数据的位置(Position)索引值，可以使用 **position()**方法得知或设定。

2. clear()、flip()与 rewind()

Buffer 的操作可以先从 **clear()**、**flip()**与 **rewind()**开始认识，当一个缓冲区刚被配置或调用 clear()方法后，limit()等于 capacity()，position()会是 0，例如配置了容量为 32 字节的 ByteBuffer 时，内部的字节数组容量、数据界限与可擦写位置分别如图 14.3 所示。

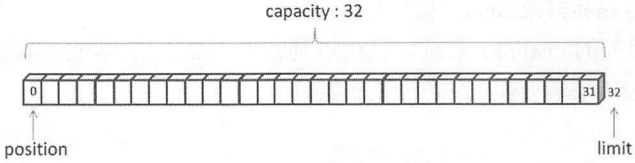


图 14.3 ByteBuffer 初建立或调用 clear()

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

若 ReadableByteChannel 对 BufferBuffer 写入了 16 字节，那么 position() 就是 16，如图 14.4 所示。

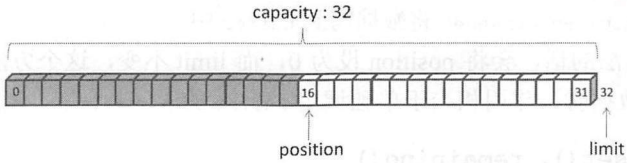


图 14.4 ByteBuffer 被写入了 16 字节

现在如果要对 ByteBuffer 中已写入的 16 字节进行读取，position 必须设为 0，为了不读取到索引 16，limit 必须设为 16。虽然可以使用 buffer.position(0).limit(16) 来完成这项任务，不过，直接调用 flip() 方法，会将 limit 值设为 position 目前值，而 position 设为 0，如图 14.5 所示。

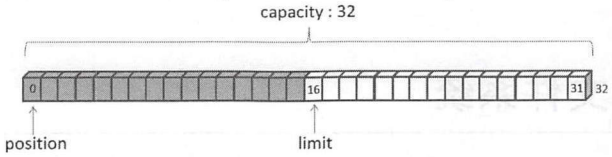


图 14.5 ByteBuffer 调用了 flip() 方法

这也就是为什么 14.1.1 节一开始的 NIO 程序示范片段会使用 flip() 方法的原因，以下是个完整的范例。

NIO NIOUtil.java

```
package cc.openhome;

import java.io.*;
import java.net.URL;
import java.nio.ByteBuffer;
import java.nio.channels.*;

public class NIOUtil {
    public static void dump(ReadableByteChannel src,WritableByteChannel dest) throws
IOException {
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        try(src; dest) {
            while(src.read(buffer) != -1) {
                buffer.flip();
                dest.write(buffer);
                buffer.clear();
            }
        }
    }

    // 测试用的 main
    public static void main(String[] args) throws Exception {
        URL url = new URL("https://openhome.cc");
        ReadableByteChannel src = Channels.newChannel(url.openStream());
        WritableByteChannel dest = new FileOutputStream("index.html").getChannel();
        NIOUtil.dump(src, dest);
    }
}
```

这个程序可以从我的网站首页下载，并自动在工作目录下存为 index.html。在 dump() 方法中，destCH.write(buffer) 会将 buffer 中从 position 至 limit 前的数据写到 WritableByteChannel

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

中，最后 position 会等于 limit，因此调用 `clear()` 将 position 设为 0，limit 设为等于容量的值，以便下个循环让 `ReadableByteChannel` 将数据写到 `buffer` 中。

调用 `rewind()` 方法的话，会将 position 设为 0，而 limit 不变，这个方法通常用在想要重复读取 `Buffer` 中某段数据时，作用相当于单独调用 `Buffer` 的 `position(0)` 方法。

3. `mark()`、`reset()`、`remaining()`

`Buffer` 上还有个 **`mark()`** 方法，可以在目前 position 上标记，在存取 `Buffer` 之后，若调用了 **`reset()`** 方法，会将 position 设回被 `mark()` 标记的位置。position 与 limit 之间为剩余可存取的资料，可以使用 **`remaining()`** 方法得知还有多少长度，使用 **`hasRemaining()`** 可以测试是否还有剩余可存取的数据。

14.2 NIO2 文件系统

JDK7 提出了 NIO2 文件系统 API，在 **`java.nio.file`**、**`java.nio.file.attribute`** 与 **`java.nio.file.spi`** 包中，提供了存取默认文件系统进行各种输入/输出的 API，既可简化现有文档输入/输出 API 的操作，也增加了许多过去没有提供的文件系统存取功能。

14.2.1 NIO2 架构

现今世界上存在着各式各样文件系统，不同文件系统会提供不同的存取方式、文件属性、权限控制等操作。在 JDK7 出现之前，常要针对特定文件系统撰写特定程序，不仅撰写方式没有标准，针对特定功能撰写程序也会增加应用程序开发者负担。

NIO2 文件系统 API 提供一组标准接口与类，应用程序开发者只要基于这些标准接口与类进行文件系统操作，底层实际如何进行文件系统操作是由文件系统提供者负责(由厂商操作)，如图 14.6 所示。

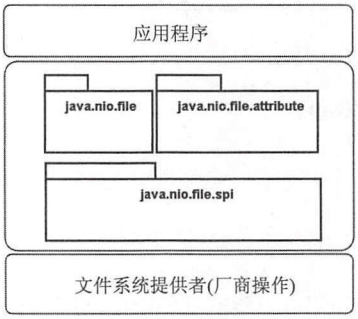


图 14.6 NIO2 文件系统 API 架构

应用程序开发者主要使用 `java.nio.file` 与 `java.nio.file.attribute`，包中必须操作的抽象类或接口由文件系统提供者操作，应用程序开发者无须担心底层实际如何存取文件系统；通常只有文件系统提供者才需关心 `java.nio.file.spi` 包。

NIO2 文件系统的中心是 **`java.nio.file.spi.FileSystemProvider`**，本身为抽象类，是文件系统提供者才要操作的类，作用是产生 `java.nio.file` 与 `java.nio.file.attribute` 中各种抽象类

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

或接口的操作对象，如图 14.7 所示。

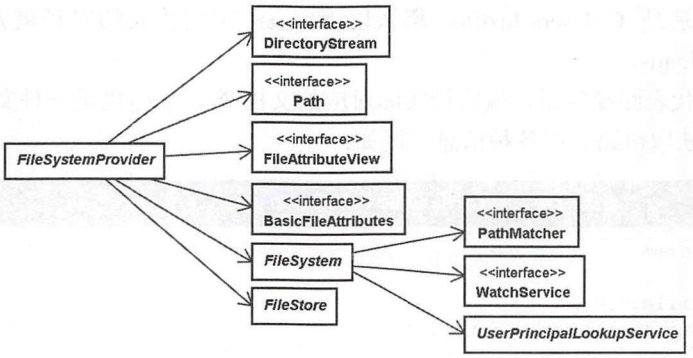


图 14.7 FileSystemProvider 产生各种操作对象

对应用程序开发者而言，只要知道有 FileSystemProvider 的存在即可。应用程序开发者可以通过 java.nio.file 包中 **FileSystems**、**Paths**、**Files** 等类提供的静态方法，取得相关操作对象或进行各种文件系统操作，这些静态方法内部会运用 FileSystemProvider 来取得所需的操作对象，完成应有的操作。

例如，想要取得 **java.nio.file.FileSystem** 操作对象，可以通过 **FileSystems.getDefault()** 取得：

```
FileSystem fileSystem = FileSystems.getDefault();
```

FileSystems.getDefault() 内部会使用 FileSystemProvider 操作对象的 getFileSystem() 方法，取得默认的 FileSystem 操作对象。如果有其他厂商操作的 FileSystemProvider 类，可以使用系统属性 java.nio.file.spi.DefaultFileSystemProvider 指定该厂商操作的类名称，这样就会使用指定的 FileSystemProvider 操作对象。

一旦更换 FileSystemProvider 操作对象，通过 FileSystems.getDefault()，就会取得该厂商的 FileSystem 操作对象。当然，**FileSystems**、**Paths**、**Files** 等类静态方法使用到的操作对象也会一并更换为该厂商的操作对象。

14.2.2 操作路径

想要操作文档，就得先指出文档路径。Path 实例是在 JVM 中路径的代表对象，也是 NIO2 文件系统 API 操作的起点，NIO2 文件系统 API 中有许多操作，都必须使用 Path 指定路径。

提示 »» 在 JDK7 出现前，路径指定是使用从 JDK1.0 开始就存在的 java.io.File，这个类功能有限，有着各平台上的行为不一致等问题，在 JDK7 中有提供 File 转换为 Path 的 API，可参考文章 *Legacy File I/O Code*：
http://docs.oracle.com/javase/tutorial/essential/io/legacy.html

想要取得 Path 实例，可以使用 **Paths.get()** 方法。最基本的使用方式，就是使用字符串路径，可使用相对路径或绝对路径。例如：

```
Path workspace = Paths.get("C:\\workspace");   \\ Windows 下绝对路径
Path books = Paths.get("Desktop\\books");      \\ Windows 下相对路径
Path path = Paths.get(args[0]);
```

Paths.get() 的第二个参数开始接受不定长度自变量，因此可指定起始路径，之后的路径分段指定。例如，以下可指定用户目录下的 Documents\Downloads：

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
Path path = Paths.get(System.getProperty("user.home"),"Documents", "Downloads");
```

如果用户目录是 C:\Users\Justin，那么以上 Path 实例代表的路径就是 C:\Users\Justin\Documents\Downloads。

Path 实例仅代表路径信息，该路径实际对应的文档或文件夹(也是一种文档)不一定存在。Path 提供一些方法取得路径的各种信息。例如：

NIO2File PathDemo.java

```
package cc.openhome;

import java.nio.file.*;
import static java.lang.System.out;

public class PathDemo {
    public static void main(String[] args) {
        Path path = Paths.get(System.getProperty("user.home"), "Documents", "Downloads");
        out.printf("toString: %s%n", path.toString());
        out.printf("getFileName: %s%n", path.getFileName());
        out.printf("getName(0): %s%n", path.getName(0));
        out.printf("getNameCount: %d%n", path.getNameCount());
        out.printf("subpath(0,2): %s%n", path.subpath(0, 2));
        out.printf("getParent: %s%n", path.getParent());
        out.printf("getRoot: %s%n", path.getRoot());
    }
}
```

路径元素计数是以文件夹为单位，最上层文件夹为索引 0。在 Windows 下的执行结果如下所示：

```
toString: C:\Users\Justin\Documents\Downloads
getFileName: Downloads
getName(0): Users
getNameCount: 4
subpath(0,2): Users\Justin
getParent: C:\Users\Justin\Documents
getRoot: C:\
```

Path 操作了 Iterable 接口，若要循序取得 Path 中分段的路径信息，也可以使用增强式 for 循环语法或 JDK8 新增的 forEach() 方法。例如：

```
Path path = Paths.get(System.getProperty("user.home"),"Documents", "Downloads");
path.forEach(out::println);
```

路径中若有冗余信息，可以使用 **normalize()** 方法移除。

例如，对于代表 C:\Users\Justin\...\Documents\Downloads 或 C:\Users\Monica\...\Justin \Documents\Downloads 的 Path 实例，以下片段都返回代表 C:\Users\Justin\Documents\Downloads 的 Path 实例：

```
Path path1 = Paths.get("C:\\Users\\Justin\\...\\Documents\\Downloads").normalize();
Path path2 = Paths.get("C:\\Users\\Monica\\...\\Justin\\Documents\\Downloads").
normalize();
```

Path 的 **toAbsolutePath()** 方法可以将相对路径 Path 转为绝对路径 Path；如果路径是符号链接(Symbolic Link)，toRealPath() 可以转换为真正的路径，若是相对路径则转换为绝对路径，若路径中有冗余信息也会移除。

路径与路径可以使用 **resolve()** 结合。例如，以下最后得到代表 C:\Users\Justin 的 Path 实例：

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
Path path1 = Paths.get("C:\\Users");
Path path2 = Paths.get("Justin");
Path path3 = path1.resolve(path2);
```

如果有两个路径，想知道如何从一个路径切换至另一个路径，则可以使用 **relativize()** 方法。例如：

```
Path p1 = Paths.get(System.getProperty("user.home"),"Documents", "Downloads");
Path p2 = Paths.get("C:\\workspace");
Path p1ToP2 = p1.relativize(p2);
System.out.println(p1ToP2);          // 显示...\...\...\...\workspace
```

可以使用 **equals()** 方法比较两个 Path 实例的路径是否相同，使用 **startsWith()** 比较路径起始是否相同，使用 **endsWith()** 比较路径结尾是否相同。如果文件系统支持符号链接，两个路径不同的 Path 实例，有可能是指向同一文档，可以使用 **Files.isSameFile()** 测试看看是否如此。

提示 »» 执行 Files.isSameFile() 时，如果两个 Path 的 equals() 是 true 就直接返回 true，不会再去确认文档是否存在。

如果想确定 Path 代表的路径，实际上是否存在文档，可以使用 **Files.exists()** 或 **Files.notExists()**。Files.exists() 仅在文档存在时返回 true，如果文档不存在或无法确认是否存在(例如没有权限存取文档)则返回 false。Files.notExists() 会在文档不存在时返回 true，如果文档存在或无法确认是否存在则返回 false。

对于文档的一些基本属性，可以使用 Files 的 **isExecutable()**、**isHidden()**、**isReadable()**、**isRegularFile()**、**isSymbolicLink()**、**isWritable()** 等方法来得知。如果需要更多文件属性信息，则必须通过 BasicFileAttributes 或搭配 FileAttributeView 来取得。

14.2.3 属性读取与设定

在过去并无标准方式取得不同文件系统支持的不同属性，在 JDK7 中，可以通过 **BasicFileAttributes**、**DosFileAttributes**、**PosixFileAttributes**，针对不同文件系统取得支持的属性信息，如图 14.8 所示。

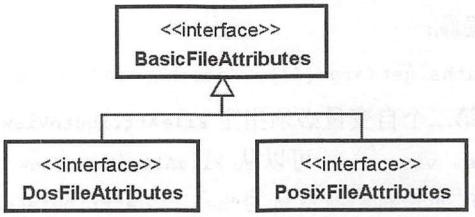


图 14.8 BasicFileAttributes 接口继承架构

BasicFileAttributes 顾名思义，就是取得各文件系统中都支持的属性，可以通过 **Files.readAttributes()** 取得 BasicFileAttributes 实例。

NIO2File BasicFileAttributesDemo.java

```
package cc.openhome;

import java.io.IOException;
import static java.lang.System.out;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

public class BasicFileAttributesDemo {
    public static void main(String[] args) throws IOException {
        Path file = Paths.get("C:\\Windows");
        BasicFileAttributes attrs = Files.readAttributes(file, BasicFileAttributes.
class);
        out.printf("creationTime: %s%n", attrs.creationTime());
        out.printf("lastAccessTime: %s%n",  attrs.lastAccessTime());
        out.printf("lastModifiedTime: %s%n", attrs.lastModifiedTime());
        out.printf("isDirectory: %b%n", attrs.isDirectory());
        out.printf("isOther: %b%n", attrs.isOther());
        out.printf("isRegularFile: %b%n",  attrs.isRegularFile());
        out.printf("isSymbolicLink: %b%n", attrs.isSymbolicLink());
        out.printf("size: %d%n", attrs.size());
    }
}
```

执行结果如下所示：

```
creationTime: 2017-09-29T08:45:11.773873Z
lastAccessTime: 2018-01-24T03:12:53.711471Z
lastModifiedTime: 2018-01-24T03:12:53.711471Z
isDirectory: true
isOther: false
isRegularFile: false
isSymbolicLink: false
size: 40960
```

creationTime()、**lastAccessTime()**、**lastModifiedTime()**返回的是 **FileTime** 实例，也可以通过 **Files.getLastModifiedTime()**取得最后修改时间；若想设定最后修改时间，可以通过 **Files.setLastModifiedTime()**指定代表修改时间的 **FileTime** 实例：

```
long currentTime = System.currentTimeMillis();
FileTime ft = FileTime.fromMillis(currentTime);
Files.setLastModifiedTime(Paths.get("C:\\workspace\\Main.java"), ft);
```

实际上，Files.setLastModifiedTime()只是个简便方法，属性设定主要可通过**Files.setAttribute()**方法。例如，配置文件为隐藏：

```
Files.setAttribute(Paths.get(args[0]), "dos:hidden", true);
```

Files.setAttribute()第二个自变量必须指定 **FileAttributeView** 子接口规范的名称，格式为**[view-name:]attribute-name**。view-name 可以从 **FileAttributeView** 子接口操作对象的 **name()**方法取得(也可查看 API 文件)，如果省略就默认为"basic"，**attribute-name** 可在 FileAttributeView 各子接口的 API 文件中查询。例如，同样设定最后修改时间，改用 Files.setAttributes()可以这样撰写：

```
long currentTime = System.currentTimeMillis();
FileTime ft = FileTime.fromMillis(currentTime);
Files.setAttribute(Paths.get("C:\\workspace\\Main.java"), "basic:lastModifiedTime", ft);
```

类似地，可以通过 **Files.getAttribute()**方法取得各种文件属性，使用方式类似 setAttributes()，也可从通过 **Files.readAttributes()**另一版本取得 Map<String, Object>对象，键部分指定属性名称，就可以取得属性值。例如：

```
Map<String, Object> attrs = Files.readAttributes(
        Paths.get(args[0]), " size,lastModifiedTime,lastAccessTime");
```

DosFileAttributes 继承 BasicFileAttributes，新增了 **isArchive()**、**isHidden()**、**isReadOnly()**、**isSystem()**等方法，可以这样取得 DosFileAttributes 实例：

```
Path file = Paths.get(args[0]);
DosFileAttributes attrs = Files.readAttributes(file, DosFileAttributes.class);
```

PosixFileAttributes 继承 BasicFileAttributes，新增了 **owner()**、**group()**方法，可取得 UserPrincipal(java.security.Principal 子接口)、GroupPrincipal (UserPrincipal 子接口)实例，可分别取得文档的群组(Group)与拥有者(Owner)信息；**permissions()**会以 Set 返回 Enum 类型的 PosixFilePermission 实例，代表文档拥有者、群组与其他用户的读/写权限信息。

提示 »» 说明 Posix 属性与权限已超出本书范围，可在网络上搜索到不少数据。如果已了解 Posix，可进一步在以下链接了解如何以 NIO2 文件系统 API 设定 Posix 相关属性：
http://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#posix

如果想取得储存装置本身的信息，可以利用 **Files.getFileStore()**方法取得指定路径的 **FileStore** 实例，或通过 FileSystem 的 **getFileStores()**方法取得所有储存装置的 FileStore 实例。以下是利用 FileStore 计算磁盘使用率的范例：

NIO2File Disk.java

```
package cc.openhome;

import java.io.IOException;
import static java.lang.System.out;
import java.nio.file.*;
import java.text.DecimalFormat;

public class Disk {
    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            FileSystem fs = FileSystems.getDefault();
            for (FileStore store: fs.getFileStores()) {
                print(store);
            }
        }
        else {
            for (String file: args) {
                FileStore store = Files.getFileStore(Paths.get(file));
                print(store);
            }
        }
    }

    public static void print(FileStore store) throws IOException {
        long total = store.getTotalSpace();
        long used = store.getTotalSpace() - store.getUnallocatedSpace();
        long usable = store.getUsableSpace();
        DecimalFormat formatter = new DecimalFormat("#,###,###");
        out.println(store.toString());
        out.printf("\t- 总容量\t%s\t 字节%n", formatter.format(total));
        out.printf("\t- 可用空间\t%s\t 字节%n", formatter.format(used));
        out.printf("\t- 已用空间\t%s\t 字节%n", formatter.format(usable));
    }
}
```

FileSystem 的 **getFileStores()**方法会以 Iterable<FileStore>返回所有储存装置的 FileStore 对象。执行结果如下所示：

```
Windows (C:)
 - 总容量      90,237,300,736   字节
 - 可用空间    77,220,610,048   字节
 - 已用空间    13,016,690,688   字节
新加卷 (D:)
 - 总容量      135,900,688,384  字节
 - 可用空间    89,746,681,856   字节
 - 已用空间    46,154,006,528   字节
新加卷 (E:)
 - 总容量      136,314,875,904  字节
 - 可用空间    15,221,354,496   字节
 - 已用空间    121,093,521,408  字节
新加卷 (F:)
 - 总容量      136,314,875,904  字节
 - 可用空间    105,042,157,568  字节
 - 已用空间    31,272,718,336   字节
```

14.2.4 操作文档与目录

如果想要删除 Path 代表的文档或目录，可以使用 **Files.delete()**方法，如果不存在，会抛出 **NoSuchFileException**，如果因目录不为空而无法删除文档，会抛出 **DirectoryNotEmptyException**。使用 **Files.deleteIfExists()**方法也可以删除文档，这个方法在文档不存在时调用，并不会抛出异常。

如果想要复制来源 Path 的文档或目录至目的地 Path，可以使用 **Files.copy()**方法，这个方法的第三个选项可以指定 **CopyOption** 接口的操作对象，CopyOption 操作类有以 Enum 类型的 **StandardCopyOption** 与 **LinkOption**。例如，指定 StandardCopyOption 的 **REPLACE_EXISTING** 实例进行复制时，若目标文档已存在就会予以覆盖，**COPY_ATTRIBUTES** 会尝试复制相关属性，LinkOption 的 **NOFOLLOW_LINKS** 则不会跟随符号链接。如下为使用 Files.copy()的范例：

```
Path srcPath = ...;
Path destPath = ...;
Files.copy(srcPath, destPath, StandardCopyOption.REPLACE_EXISTING);
```

Files.copy()还有两个重载版本，一个是接受 InputStream 作为来源，可直接读取数据，并将结果复制至指定的 Path 中；另一个 Files.copy()版本是将来源 Path 复制至指定的 OutputStream。例如，以下为改写 10.1.1 节中的 Download：

NIO2File Download.java

```
package cc.openhome;

import java.io.*;
import java.net.URL;
import java.nio.file.*;
import static java.nio.file.StandardCopyOption.*;

public class Download {
    public static void main(String[] args) throws IOException {
        URL url = new URL(args[0]);
```

```
            Files.copy(url.openStream(), Paths.get(args[1]), REPLACE_EXISTING);
        }
    }
```

若要进行文档或目录移动，可以使用 **Files.move()**方法，使用方式与 Files.copy()方法类似，可指定来源 Path、目的地 Path 与 CopyOption。如果文件系统支持原子移动，可在移动时指定 **StandardCopyOption.ATOMIC_MOVE** 选项。

如果要建立目录，可以使用 **Files.createDirectory()**方法，如果调用时父目录不存在，会抛出 **NoSuchFileException**。**Files.createDirectories()**会在父目录不存在时一并建立。

提示》》 可在建立目录时一并使用 FileAttribute 指定文件属性，例如在支持 Posix 的文件系统上建立目录：

```
Set<PosixFilePermission> perms =
        PosixFilePermissions.fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attr =
        PosixFilePermissions.asFileAttribute(perms);
Files.createDirectory(file, attr);
```

如果要建立暂存目录，可以使用 Files.createTempDirectory()方法，这个方法有可指定路径与使用默认路径建立暂存目录两个版本。

在第 10 章介绍过基本输入/输出，对于 java.io 中的基本输入/输出 API，NIO2 也做了封装。例如，如果 Path 实例是个文档，可使用 **Files.readAllBytes()**读取整个文档，然后以 byte[]返回文档内容；如果文档内容都是字符，则可使用 **Files.readAllLines()**指定文档 Path 与编码，读取整个文档，将文档中每行收集在 List<String>中返回。

在第 12 章中介绍过，JDK8 新增了 Stream API，在 NIO2 的文档读取上，Files 新增了 **lines()**静态方法，它返回的是 Stream<String>，适用于需要管线化、惰性操作的场合，**lines()**内部会打开文档，不使用时需要调用 **close()**方法来释放资源，返回的 Stream 可以搭配 JDK7 尝试关闭(Try-with-resources)资源语法来关闭。例如：

```
try (Stream<String> lines = Files.lines(Paths.get(args[0]))) {
    lines.forEach(out::println);
}
```

如果文档内容都是字符，则需要在读取或写入时使用缓冲区，也可以使用 **Files.newBufferedReader()**、**Files.newBufferedWriter()**指定文档 Path 与编码，它们分别会返回 BufferedReader、BufferedWriter 实例，可以使用它们来进行文档读取或写入。例如，如果本来有个建立 BufferedReader 的片段如下：

```
BufferedReader reader =
    new BufferedReader(
        new InputStreamReader(
            new FileInputStream(args[0]), "UTF-8"));
```

可使用 Files.newBufferedReader()改写成如下：

```
BufferedReader reader =
    Files.newBufferedReader(Path.get(args[0]), "UTF-8");
```

在使用 Files.newBufferedWriter()时，还可以指定 **OpenOption** 接口的操作对象，其操作类为 **StandardOpenOption** 与 LinkOption(操作了 CopyOption 与 OpenOption)，可以指定开启文档时

的行为，可以查看 StandardOpenOption 与 LinkOption 的 API，了解有哪些选项可以使用。

如果想要以 InputStream、OutputStream 处理文档，也有对应的 **Files.newInputStream()**、**Files.newOutputStream()**可以使用。

14.2.5 读取、访问目录

如果想要取得文件系统根目录路径信息，可以使用 FileSystem 的 **getRootDirectories()**方法，这会取回 Iterable<String>对象，可用增强式 for 循环或 JDK8 新增的 forEach()方法取得根目录路径信息。例如：

NIO2File Roots.java

```
package cc.openhome;

import static java.lang.System.out;
import java.nio.file.*;

public class Roots {
    public static void main(String[] args) {
        Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
        dirs.forEach(out::println);
    }
}
```

Windows 下的执行结果如下：

```
C:\
D:\
```

也可以使用 **Files.newDirectoryStream()**方法取得 **DirectoryStream** 接口操作对象，代表指定路径下的所有文档。

在不使用 DirectoryStream 对象时必须用 close()方法关闭相关资源，DirectoryStream 继承了 Closeable 接口，其父接口为 AutoClosable 接口，所以可搭配尝试关闭资源语法来简化程序撰写。Files.newDirectoryStream()实际返回的是 DirectoryStream<Path>，由于 Directory Stream 也继承了 Iterable 接口，所以可使用增强式 for 循环语句或 JDK8 新增的 forEach()方法来逐一取得 Path。

下面这个范例可以从命令行自变量指定目录路径，查询出该目录下的文档。

NIO2File Dir.java

```
package cc.openhome;

import java.io.IOException;
import static java.lang.System.out;
import java.nio.file.*;
import java.util.*;

public class Dir {
    public static void main(String[] args) throws IOException {
        try(DirectoryStream<Path> directoryStream =
                        Files.newDirectoryStream(Paths.get(args[0]))) {
            List<String> files = new ArrayList<>();
```

```
            for(Path path : directoryStream) {
                if(Files.isDirectory(path)) {
                    out.printf("[%s]%n", path.getFileName());
                }
                else {
                    files.add(path.getFileName().toString());
                }
            }
            files.forEach(out::println);
        }
    }
}
```

这个范例会先列出目录，再列出文档，如果命令行自变量指定的是 C:\，则执行结果如下：

```
[$Recycle.Bin]
[Boot]
[Config.Msi]
[Documents and Settings]
[farston]
[Greenware]
[Intel]
[MSOCache]
[PerfLogs]
[Program Files]
[Program Files (x86)]
[ProgramData]
[Recovery]
[System Volume Information]
[Users]
[Windows]
[Winware]
[workspace]
bootmgr
BOOTNXT
devlist.txt
farstone_pe.letter
Finish.log
hiberfil.sys
pagefile.sys
swapfile.sys
```

如果想要访问目录中所有文档与子目录，可以操作 **FileVisitor** 接口，其中定义了 4 个必须操作的方法：

```
package java.nio.file;
import java.nio.file.attribute.BasicFileAttributes;
import java.io.IOException;
public interface FileVisitor<T> {
    FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
        throws IOException;
    FileVisitResult visitFile(T file, BasicFileAttributes attrs)
        throws IOException;
    FileVisitResult visitFileFailed(T file, IOException exc)
        throws IOException;
    FileVisitResult postVisitDirectory(T dir, IOException exc)
        throws IOException;
}
```

从指定的目录路径开始，每次要开始访问该目录内容前，会调用 **preVisitDirectory()**，要访问文档时会调用 **visitFile()**，访问文档失败会调用 **visitFileFailed()**，访问整个目录内容后会调用 **postVisitDirectory()**。如果有多层目录，图 14.9 显示了访问时调用方法的顺序。

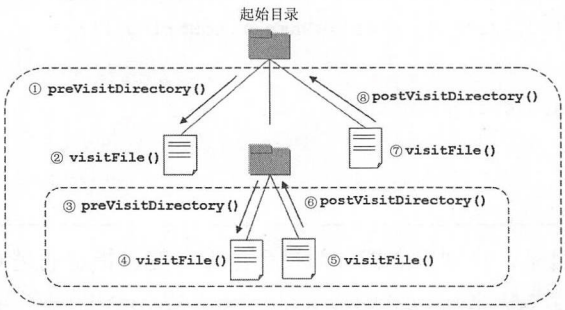


图 14.9 FileVisitor 方法调用顺序

如果只对 FileVisitor 其中一两个方法有兴趣，可以继承 **SimpleFileVisitor** 类，这个类操作了 FileVisitor 接口，只要继承之后重新定义感兴趣的方法就可以了。例如：

NIO2File ConsoleFileVisitor.java

```
package cc.openhome;

import java.io.IOException;
import static java.lang.System.*;
import java.nio.file.*;
import static java.nio.file.FileVisitResult.*;
import java.nio.file.attribute.*;

public class ConsoleFileVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult preVisitDirectory(Path path,BasicFileAttributes attrs)
                           throws IOException {
        printSpace(path);
        out.printf("[%s]%n", path.getFileName());
        return CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path path, BasicFileAttributes attr) {
        printSpace(path);
        out.printf("%s%n", path.getFileName());
        return CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc) {
        err.println(exc);
        return CONTINUE;
    }

    private void printSpace(Path path) {
```

```
            out.printf("%" + path.getNameCount() * 2 + "s", "");
        }
    }
```

preVisitDirectory()、visitFile()、visitFileFailed()等方法，必须返回 **FileVisitorResult**，返回 FileVisitorResult.CONTINUE 表示继续访问目录。如果要使用 FileVisitor 访问目录，可以使用 **Files.walkFileTree()**方法。例如：

NIO2File DirAll.java

```
package cc.openhome;

import java.io.IOException;
import java.nio.file.*;

public class DirAll {
    public static void main(String[] args) throws IOException {
        Files.walkFileTree(Paths.get(args[0]), new ConsoleFileVisitor());
    }
}
```

执行的结果如下：

```
[workspace]
    [NIO2]
      [build]
        [modules]
          [cc.openhome]
            .netbeans_automatic_build
            .netbeans_update_resources
            [cc]
              [openhome]
                BasicFileAttributesDemo.class
                ConsoleFileVisitor.class
                Dir.class
                DirAll.class
                ...
```

在第 12 章中介绍过，JDK8 增加了 Stream API，在 NIO2 的目录访问上，Files 新增了 **list()** 与 **walk()**静态方法，它返回的是 Stream<Path>，适用于需要管线化、惰性操作的场合，list()会列出当前目录下所有文档，walk()会列出当前目录及子目录下所有文档，与前面介绍的 lines()一样，**list()**与 **walk()**返回的 **Stream** 不使用时需要调用 **close()**方法来释放资源，可以搭配 JDK7 尝试关闭(Try-with-resources)资源语法来关闭。例如：

```
try (Stream<Path> paths = Files.list(Paths.get(args[0]))) {
    paths.forEach(out::println);
}

try (Stream<Path> paths = Files.walk(Paths.get(args[0]))) {
    paths.forEach(out::println);
}
```

提示 »» 如果对监控目录变化有兴趣，NIO2 提供了 WatchService 等低层次的 API，可参考以下网址(*Watching a Directory for Changes*)了解进一步信息：
http://docs.oracle.com/javase/tutorial/essential/io/notification.html

14.2.6 过滤、搜索文档

如果在列出目录内容时过滤想显示的文档，例如只想显示.class 与.jar 文档，可以在使用 **Files.newDirectoryStream()** 时的第二个参数指定过滤条件为*.{class,jar}。例如：

```
try(DirectoryStream<Path> directoryStream = Files.newDirectoryStream(
                                    Paths.get(args[0]), "*.{class,jar}")) {
        directoryStream.forEach(path -> out.println(path.getFileName()));
}
```

像*.{class,jar}这样的语法称为 Glob，是一种模式比较语法，比规则表示式简单，常用于目录与文件名的比较。Glob 语法使用符号与说明，如表 14.1 所示。

表 14.1 Glob 语法使用符号与说明

符 号	说 明
*	比较零个或多个字符
**	跨目录比较零个或多个字符
?	比较一个字符
{}	比较收集的任一子模式，例如{class,jar}比较 class 或 jar，{tmp,temp*}比较 tmp 或 temp 开头
[]	比较收集的任一字符，例如[acx]比较 a、c、x 任一字符，可使用-比较范围，例如[a-z]比较 a 到 z 任一字符，[A-Z,0-9]比较 A 到 Z 或 0 到 9 任一字符，在[]中的*、?、\就是进行字符比较，例如[*?\]就是比较*、?、\任一字符
\	忽略符号，例如要比较*、?、\，就要撰写为*、\?、\\
其他字符	比较字符本身

以下是几个 Glob 比较范例：

- *.java 比较.java 结尾的字符串。
- **/*Test.java 跨目录比较 Test.java 结尾的字符串，例如 BookmarkTest.java、CommandTest.java 都符合。
- ???符合三个字符，例如 123、abc 会符合。
- a?*.java 比较 a 之后至少一个字符，并以.java 结尾的字符串。
- *.{class,jar}符合.class 或.jar 结尾的字符串。
- *[0-9]*比较的字符串中要有一个数字。
- {*[0-9]*,*.java}比较字符串中要有一个数字，或者是.java 结尾。

以下范例，可指定 Glob 搜索工作目录下符合的文档。

NIO2File Ls.java

```
package cc.openhome;

import java.io.IOException;
import static java.lang.System.out;
import java.nio.file.*;
```

```
public class LS {
    public static void main(String[] args) throws IOException {
        // 默认取得所有文档
        String glob = args.length == 0 ? "*" : args[0];

        // 取得目前工作路径
        Path userPath = Paths.get(System.getProperty("user.dir"));
        try (DirectoryStream<Path> directoryStream =
                Files.newDirectoryStream(userPath, glob)) {
            directoryStream.forEach(
                path -> out.println(path.getFileName()));
        }
    }
}
```

如果启动 JVM 时指定命令行自变量为 build*，表示使用 Glob 语法为 `build*`，那么工作目录下所有 build 开头的文档或目录会显示出来。例如，在 NetBeans 中执行时指定命令行自变量为 build*会显示：

```
build
build.xml
```

`Files.newDirectoryStream()`的另一版本接受 **`DirectoryStream.Filter`** 接口操作对象，如果 Glob 语法无法满足条件过滤需求时，可以自行操作 `DirectoryStream.Filter` 的 **`accept()`** 方法自定义过滤条件(例如采用更强大的规则表示式)，`accept()`方法返回 `true` 表示符合过滤条件。例如只过滤出目录：

```
DirectoryStream.Filter<Path> filter = new DirectoryStream.Filter<Path>() {
    public boolean accept(Path path) throws IOException {
        return (Files.isDirectory(path));
    }
};
try (DirectoryStream<Path> directoryStream =
     Files.newDirectoryStream(Paths.get(args[0]), filter)) {
    directoryStream.forEach(path -> out.println(path.getFileName()));
}
```

也可以使用 `FileSystem` 实例的 **`getPathMatcher()`** 取得 **`PathMatcher`** 接口操作对象，在取得 `PathMatcher` 时可以指定使用哪种模式比较语法，**`"regex"`**表示使用规则表示式语法，`"glob"`表示使用 Glob 语法(其他厂商操作也许还会提供不同语法)。例如：

```
PathMatcher matcher = FileSystems.getDefault()
                                 .getPathMatcher("glob:*.{class,jar}");
```

取得 `PathMatcher` 后，可以使用 **`matches()`** 方法进行路径比较，返回 `true` 表示符合模式。例如改写上一个范例，可以指定使用规则表示式或 Glob。

NIO2File Ls2.java

```
package cc.openhome;
package cc.openhome;

import java.io.IOException;
import static java.lang.System.out;
import java.nio.file.*;
```

```
public class LS2 {
    public static void main(String[] args) throws IOException {
        // 默认使用 Glob 取得所有文档
        String syntax = args.length == 2 ? args[0] : "glob";
        String pattern = args.length == 2 ? args[1] : "*";
        out.println(syntax + ":" + pattern);

        // 取得目前工作路径
        Path userPath = Paths.get(System.getProperty("user.dir"));
        PathMatcher matcher = FileSystems.getDefault()
                              .getPathMatcher(syntax + ":" + pattern);
        try (DirectoryStream<Path> directoryStream =
               Files.newDirectoryStream(userPath)) {
           directoryStream.forEach(path -> {
               Path file = Paths.get(path.getFileName().toString());
               if(matcher.matches(file)) {
                   out.println(file.getFileName());
               }
           });
        }
    }
}
```

注意»» 如果 path 参考至 Path 实例，使用 Files.newDirectoryStream() 指定 Glob 比较时，比较的字符串对象是 path.getFileName().toString()；使用 PathMatcher 的 matches() 时，比较的字符串对象是 path.toString()。

14.3 重点复习

NIO 使用频道(Channel)来衔接数据节点，在处理数据时，NIO 可以设定缓冲区(Buffer)容量，在缓冲区中对感兴趣的数据区块进行标记，如标记读取位置、数据有效位置，对于这些区块标记，提供了 clear()、rewind()、flip()、compact() 等高级操作。

想要取得 Channel 的操作对象，可以使用 Channels 类，它定义了静态方法 newChannel()，可以从 InputStream、OutputStream 分别建立 ReadableByteChannel、WritableByteChannel，有些 InputStream、OutputStream 实例本身也有方法可以取得 Channel 实例。举例来说，FileInputSteam、FileOutputStream 都有个 getChannel() 方法可以分别取得 FileChannel 实例。如果已经有相关的 Channel 实例，也可以通过 Channels 上其他 newXXX() 静态方法，取得 InputStream、OutputStream、Reader、Writer 实例。

Buffer 的直接子类们都有一个 allocate() 静态方法，可以指定 Buffer 容量(Capacity)。Buffer 是个容器，填装的数据不能超过它的容量，Buffer 的容量大小可以使用 capacity() 方法取得。实际可读取或写入的数据界限(Limit)索引值可以由 limit() 方法得知或设定，下一个可读取数据的位置(Position)索引值可以使用 position() 方法得知或设定。

NIO2 文件系统 API 提供一组标准接口与类，应用程序开发者只要基于这些标准接口与类进行文件系统操作，底层实际如何进行文件系统操作，是由文件系统提供者负责(由厂商操作)。

应用程序开发者可以通过 java.nio.file 包中 FileSystems、Paths、Files 等类提供的静态方法，取得相关操作对象或进行各种文件系统操作，这些静态方法内部会运用 FileSystemProvider

来取得所需的操作对象，完成应有的操作。

14.4 课后练习

撰写程序可如图 14.10 所示指定目录与 Glob 模式，递归搜索指定目录与子目录中符合模式的文件名，Glob 模式必须包括在""中以避免被控制台解释为特定字符(例如，通配符'*')。

```
C:\workspace>java cc.openhome.Exercise "C:\Program Files" -name "*.zip"
C:\Program Files\astah-community\jre\lib\deploy\ffjcext.zip
C:\Program Files\Intel\WiFi\bin\DualServer.zip
C:\Program Files\Java\jdk1.7.0\jre\lib\deploy\ffjcext.zip
C:\Program Files\Java\jdk1.7.0\src.zip
C:\Program Files\Java\jdk1.8.0\javafx-src.zip
C:\Program Files\Java\jdk1.8.0\jre\lib\deploy\ffjcext.zip
C:\Program Files\Java\jdk1.8.0\src.zip
```

图 14.10　指定目录与 Glob 模式

Chapter 15 通用 API

学习目标

- 使用日志 API
- 了解国际化基础
- 运用规则表示式
- 处理数字
- 访问堆栈追踪

15.1 日志

系统中有许多值得记录的信息，例如捕捉异常之后，有些异常值得显示给用户观看就抛出。而对于开发人员或系统人员才有意义的异常，可以记录下来，那么该记录哪些信息(时间、信息产生处等)？用何种方式记录(文档、数据库、远程主机等)？记录格式(控制台、纯文本、XML等)？这些都是在记录时值得考虑的要素。Java SE提供了日志(Logging)API，可以基于标准调用使用。

15.1.1 日志API简介

java.util.logging包提供了日志功能相关类与接口，它们是从JDK1.4之后加入标准API，尽管功能上比不上第三方日志链接库，然而不必额外配置日志组件，就可在标准Java平台很好地使用。

在JDK9支持模块化之后，java.util.logging包被划分至**java.logging**模块，如果采取模块化方式编写程序，必须在模块描述文件中加入requires java.logging:

Logging module-info.java

```
module cc.openhome {
    requires java.logging;
}
```

使用日志的起点是Logger类，Logger实例的创建有许多要处理的要素(如前面简介提供的几个要素，还有稍后会提到的名称空间处理问题等)，Logger类的构造函数标示为protected，不是java.util.logging同包的类不能直接以new创建，要获取Logger实例，必须使用Logger的静态方法getLogger()。例如：

```
Logger logger = Logger.getLogger("cc.openhome.Main");
```

调用getLogger()时，必须指定Logger实例所属名称空间(Name space)，名称空间以“.”作为层级区分，名称空间层级相同的Logger，其父Logger组态相同。例如，若有个Logger名称空间为cc.openhome，则名称空间cc.openhome.Some与cc.openhome.Other的Logger，其父Logger组态都是cc.openhome名称空间的Logger组态。

通常在哪个类中获取的Logger，名称空间就会命名为哪个类全名(像上例就是在cc.openhome.Main中获取Logger)。经常地，也会通过以下方式获取Logger:

```
Logger logger = Logger.getLogger(Main.class.getName());
```

之后在反射(Reflection)中会介绍，类之后接下.class，可以获取该类的java.lang.Class实例，调用其getName()就可以获取类全名。获取Logger实例之后，可以使用log()方法输出信息，输出信息时可以使用Level的静态成员指定信息层级(Level)。例如：

Logging LoggerDemo.java

```
package cc.openhome;

import java.util.logging.*;
```

```
public class LoggerDemo {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger(LoggerDemo.class.getName());
        logger.log(Level.WARNING, "WARNING 信息");
        logger.log(Level.INFO, "INFO 信息");
        logger.log(Level.CONFIG, "CONFIG 信息");
        logger.log(Level.FINE, "FINE 信息");
    }
}
```

执行结果如下：

```
1 月 31, 2018 5:21:42 下午 cc.openhome.LoggerDemo main
警告: WARNING 信息
1 月 31, 2018 5:21:42 下午 cc.openhome.LoggerDemo main
信息: INFO 信息
```

可以看到，除了指定的信息之外，默认的 Logger 还会记录时间、类、方法等信息。咦？怎么只看到 Level.WARNING 与 Level.INFO 的信息？为何会默认输出至控制台？若要将信息输出至文档怎么办？想改变信息的输出格式呢？除了日志层级指定之外，如果要依额外条件决定是否输出信息呢？要了解这一切的答案，必须先了解日志 API 各类与接口间的调用关系，如图 15.1 所示。

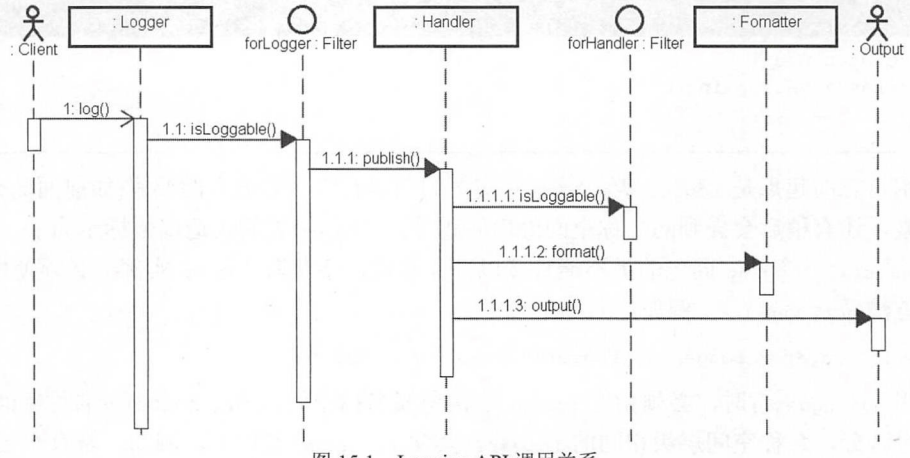


图 15.1　Logging API 调用关系

图 15.1 是简化后的 API 调用关系图，如果客户端调用了 Logger 实例的 log() 方法，首先会依据 Level 过滤信息，再看看 Logger 有无设定 Filter 接口的实例，如果有且其 isLoggable() 返回 true，才会调用 Handler 实例的 publish() 方法，Handler 也可设定自己的 Filter 实例，如果有且其 isLoggable() 返回 true，就调用 Formatter 实例的 format() 方法格式化信息，最后才调用输出对象将格式化后的信息输出。

简单来说，Logger 是记录信息的起点，要输出的信息必须先通过 Logger 的 Level 与 Filter 过滤，再通过 Handler 的 Level 与 Filter 过滤，格式化信息的动作交给 Formatter，输出信息的动作实际上是 Handler 负责。

还得知道的是，正如前面介绍过的，Logger 有层级关系，名称空间层级相同的 Logger，父 Logger 组态会相同，每个 Logger 处理完自己的日志动作后，会向父 Logger 传播，让父 Logger 也可以处理日志。

15.1.2 指定日志层级

Logger 与 Handler 默认都会先依据 Level 过滤信息，如果没有做任何修改，获取的 Logger 实例之父 Logger 组态，就是 **Logger.GLOBAL_LOGGER_NAME** 名称空间 Logger 实例的组态，这个实例的 Level 设定为 INFO，可通过 Logger 实例的 **getParent()** 获取父 Logger 实例，可通过 **getLevel()** 获取设定的 Level 实例。例如：

```
jshell> import java.util.logging.*;

jshell> Logger logger = Logger.getLogger("cc.openhome.Some");
logger ==> java.util.logging.Logger@3d99d22e

jshell> Logger global = Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);
global ==> java.util.logging.Logger@cd2dae5

jshell> logger.getLevel();
$4 ==> null

jshell> logger.getParent().getLevel();
$5 ==> INFO

jshell> global.getParent().getLevel();
$6 ==> INFO
```

Logger 的信息处理会往父 Logger 传播，也就是说，在没有做任何组态设定的情况下，默认获取的 **Logger** 实例，层级必须大于或等于 **Logger.GLOBAL_LOGGER_NAME** 名称空间 **Logger** 实例设定的 **Level.INFO**，才有可能输出信息。可以通过 Logger 的 **setLevel()** 指定 Level 实例，使用 Level 内建的几个静态成员来指定：

```
Level.OFF(Integer.MAX_VALUE)
Level.SEVERE(1000)
Level.WARNING(900)
Level.INFO(800)
Level.CONFIG(700)
Level.FINE(500)
Level.FINER(400)
Level.FINEST(300)
Level.ALL(Integer.MIN_VALUE)
```

这些静态成员都是 Level 的实例，可以使用 intValue() 获取内含 int 值，Logger 本身可以通过 setLevel() 设定 Level 实例。若 log() 时指定的 Level 实例内含的 int 值小于 Logger 设定的 Level 实例内含的 int 值，Logger 就不会记录信息，也因此 Level.OFF 会用于关闭所有信息输出，Level.ALL 会用于允许所有信息输出。

提示>>> 必要时也可使用 new 方式创建 Level 实例，在过滤信息时也是依据内含 int 值来判断要不要输出信息。

在经过 Logger 过滤之后，还得再经过 Handler 的过滤，一个 Logger 可以拥有多个 Handler，可通过 Logger 的 addHandler() 新增 Handler 实例。前面谈过，每个 Logger 处理完自己的日志动

作后，会向父 Logger 传播，让父 Logger 也可以处理日志，所以实际上进行信息输出时，目前 Logger 的 Handler 处理完，还会传播给父 Logger 的所有 Handler 处理(在通过父 Logger 层级的情况下)。通过 getHandlers()方法可获取目前已有的 Handler 实例数组。例如：

```
Logger logger = Logger.getLogger(Some.class.getName());
out.println(logger.getHandlers().length); // 显示 0，表示没有 Handler
// 以下会显示两行，一行包括 java.util.logging.ConsoleHandler 字样
// 一行包括 INFO 字样
for(Handler handler : logger.getParent().getHandlers()) {
    System.out.println(handler);
    System.out.println(handler.getLevel());
}
```

也就是说，在没有做任何组态设定的情况下，获取的 Logger 实例，只会使用 Logger.GLOBAL_LOGGER_NAME 名称空间 Logger 实例拥有的 Handler，默认是使用 ConsoleHandler，为 Handler 的子类，作用是在控制台下输出日志信息，默认的层级是 Level.INFO。

Handler 可通过 setLevel()设定信息，一个信息要经过 Logger 与 Handler 的过滤后才可输出，所以 15.1.1 节的范例，若要显示 INFO 以下的信息，不仅要将 Logger 的层级设定为 Level.INFO，也得将 Handler 的层级设定为 Level.INFO。例如：

Logging LoggerDemo2.java

```
package cc.openhome;

import java.util.logging.*;

public class LoggerDemo2 {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger(LoggerDemo2.class.getName());
        logger.setLevel(Level.FINE);
        for(Handler handler : logger.getParent().getHandlers()) {
            handler.setLevel(Level.FINE);
        }
        logger.log(Level.WARNING, "WARNING 信息");
        logger.log(Level.INFO, "INFO 信息");
        logger.log(Level.CONFIG, "CONFIG 信息");
        logger.log(Level.FINE, "FINE 信息");
    }
}
```

执行结果如下，将输出程序中所有指定的信息：

```
1 月 31, 2018 5:24:42 下午 cc.openhome.LoggerDemo2 main
警告: WARNING 信息
1 月 31, 2018 5:24:42 下午 cc.openhome.LoggerDemo2 main
信息: INFO 信息
1 月 31, 2018 5:24:42 下午 cc.openhome.LoggerDemo2 main
状态: CONFIG 信息
1 月 31, 2018 5:24:42 下午 cc.openhome.LoggerDemo2 main
详细: FINE 信息
```

对于一些日志层级，Logger 实例有对应的简便方法，如 severe()、warning()、info()、config()、fine()、finer()、finest()，JDK8 引入了 Lambda 之后，这些方法也多了重载版本，

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

可以接受 Supplier 实例。如果你的日志动作是比较消耗资源的话，就可以如下撰写，这样层级不到的时候，就不会执行 expansiveLogging()：

```
logger.debug(() -> expansiveLogging());
```

15.1.3 使用 Handler 与 Formatter

负责日志输出的是 Handler 实例，标准 API 中提供了几个 Handler 操作类，继承架构如图 15.2 所示。

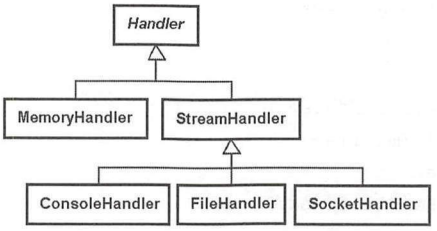


图 15.2 Handler 继承架构

MemoryHandler 不会格式化日志信息，信息会暂存于内存缓冲区，直到超过缓冲区大小，才将信息输出至指定的目标 Handler。StreamHandler 可自行指定信息输出时使用的 OutputStream 实例，它与子类都会使用指定的 Formatter 格式化信息。ConsoleHandler 创建时，会自动指定 OutputStream 为 System.err，所以日志信息会显示在控制台。FileHandler 创建时会建立日志输出时使用的 FileOutputStream，文档位置与名称可以使用模式(Pattern)字符串指定。SocketHandler 创建时可以指定主机位置与端口，内部将自动建立网络联机，将日志信息传送至指定的主机。

Logger 可以使用 addHandler()新增 Handler 实例，使用 removeHandler()移除 Handler。下面范例将目前 Logger 与新建的 FileHandler 层级设定 Level.CONFIG，并使用 addHandler()设定至 Logger 实例。

Logging HandlerDemo.java

```
package cc.openhome;

import java.io.IOException;
import java.util.logging.*;

public class HandlerDemo {
    public static void main(String[] args) throws IOException {
        Logger logger = Logger.getLogger(HandlerDemo.class.getName());
        logger.setLevel(Level.CONFIG);
        FileHandler handler = new FileHandler("%h/config.log");
        handler.setLevel(Level.CONFIG);
        logger.addHandler(handler);
        logger.config("Logger 组态完成");
    }
}
```

在建立 FileHandler 指定模式字符串时，可以使用"%h"来表示用户的根(home)目录(Windows 用户根目录是在 C:\Documents and Settings\用户名称)，还可以使用"%t"获取系统暂存目录，或

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

者使用"%g"自动为文档编号，例如设定为"%h/config%g.log"，表示将 configN.log 文件储存在用户根目录，N 表示每个信息的文档编号，会自动递增。

Logger 的 config()是个简便方法，可以直接以 Level.CONFIG 层级输出信息，另外也有 severe()、info()等简便方法。上面这个范例只会在目前 Logger 增加 FileHandler，因为父 Logger 默认层级为 LEVEL.INFO，信息不会再显示在控制台，而会储存在用户根目录的 config.log 中，默认会以 XML 格式储存。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2018-02-07T14:21:51</date>
  <millis>1399443711854</millis>
  <sequence>0</sequence>
  <logger>cc.openhome.HandlerDemo</logger>
  <level>CONFIG</level>
  <class>cc.openhome.HandlerDemo</class>
  <method>main</method>
  <thread>1</thread>
  <message>Logger 组态完成</message>
</record>
</log>
```

这是因为 FileHandler 默认的 Formatter 是 XMLFormatter，前面看过的 ConsoleHandler 默认则使用 SimpleFormatter，这两个类是 Formatter 的子类，可以通过 Handler 的 setFormatter()方法设定 Formatter。

事实上，如果不想让父 Logger 的 Handler 处理日志，可以调用 Logger 实例的 setUseParentHandlers()设定为 false，这样日志信息就不会传播给父 Logger。也可以使用 Logger 实例的 setParent()方法指定父 Logger。

提示>>> 如果要以特定编码输出信息或储存文档，Handler 有个 setEncoding()方法，可以指定文字编码。

15.1.4 自定义 Handler、Formatter 与 Filter

如果 java.util.logging 包中提供的 Handler 成果都不符合需求，可以继承 Handler 类，操作抽象方法 **publish()**、**flush()**与 **close()**方法来自定义 Handler，建议操作时考虑信息过滤与格式化。如以下操作流程：

```
...
public class CustomHandler extends Handler {
        ...
    public void publish(LogRecord logRecord) {
        if (!isLoggable(record)) {
            return;
        }
        String logMsg = getFormatter().format(logRecord);
        out.write(logMsg); // out 是输出目的地对象
    }
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        public void flush() {
            ...出清信息
        }

        public void close() {
            ...关闭输出对象
        }
    }
```

操作时要记得，在职责分配上，Handler是负责输出，格式化交由Formatter，而信息过滤交由Filter。Handler有默认的isLoggable()操作，会先依据Level过滤信息，再使用指定的Filter过滤信息：

```
    ...
        public boolean isLoggable(LogRecord record) {
            int levelValue = getLevel().intValue();
            if (record.getLevel().intValue() < levelValue ||
                levelValue == offValue) {
                return false;
            }
            Filter filter = getFilter();
            if (filter == null) {
                return true;
            }
            return filter.isLoggable(record);
        }
    ...
```

如果要自定义Formatter，可以继承Formatter后操作抽象方法format()，这个方法会传入LogRecord，储存所有日志信息。例如，将ConsoleHandler的Formatter设定为自定义的Formatter：

Logging FormatterDemo.java

```
package cc.openhome;

import java.util.Date;
import java.util.logging.*;

public class FormatterDemo {

    public static void main(String[] args) {
        Logger logger = Logger.getLogger(FormatterDemo.class.getName());
        logger.setLevel(Level.CONFIG);
        ConsoleHandler handler = new ConsoleHandler();
        handler.setLevel(Level.CONFIG);
        handler.setFormatter(new Formatter() {
            @Override
            public String format(LogRecord record) {
                return "日志来自 " + record.getSourceClassName()
                                   + record.getSourceMethodName() + "\n"
                       + "\t层级\t: " + record.getLevel() + "\n"
                       + "\t信息\t: " + record.getMessage() + "\n"
                       + "\t时间\t: "  + new Date(record.getMillis())
                       + "\n";
            }
        });
        logger.addHandler(handler);
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        logger.config("自定义 Formatter 信息");
    }
}
```

执行结果如下所示：

```
日志来自 cc.openhome.FormatterDemo.main
        层级  : CONFIG
        信息  : 自定义 Formatter 信息
        时间  : 2018-02-07T06:25:37.082Z
```

Logger 与 Handler 默认只会依据层级过滤信息，Logger 与 Handler 都有 setFilter() 方法，可以指定 Filter 操作对象。如果想让 Logger 与 Handler 除了依据层级过滤之外，还可以加入额外过滤条件，就可以操作 Filter 接口：

```
package java.util.logging;
public interface Filter {
    public boolean isLoggable(LogRecord record);
}
```

提示 »» 在 Java 中，有许多日志链接库选择，例如 Log4j、Apache Common Logging、SL4J 或是 Log4j2 等。若读者想了解这些链接库的基本差异，可以从《哪来这么多日志链接库》开始，链接地址如下：
openhome.cc/Gossip/Programmer/Logging.html

15.1.5 使用 logging.properties

以上都是使用程序撰写方式，改变 Logger 对象的组态。实际上，可以通过 logging.properties 来设定 Logger 组态，这很方便。例如程序开发阶段，在.properties 中设定所有 Level.WARNING 层级的信息输出，在程序上线之后，若想关闭不会影响程序运行的警告日志，以减少程序不必要的输出(不必要的日志输出会影响程序运行效率)，只要在.properties 中做个修改即可。

在 Java SE9 的 JDK 或 JRE 目录的 conf 目录中，有个 logging.properties 文档，是设定 Logger 组态的参考范例：

```
############################################################
#  全局 Logger 组态
############################################################

# "handlers" 可以逗号分隔指定多个 Handler 类
# 在 JVM 启动后会完成 Handler 设定，指定的类必须在 CLASSPATH 中
# 默认是 ConsoleHandler
handlers= java.util.logging.ConsoleHandler

# 下面是同时设定 FileHandler 与 ConsoleHandler 的范例
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# 全局 Logger 默认层级(不是 Handler 默认层级)
# 默认是 INFO
.level= INFO

############################################################
```

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
# Handler 默认组态
############################################################

# FileHandler 默认组态
# Formatter 默认是 XMLFormatter
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# ConsoleHandler 默认组态
# 层级默认是 INFO
# Formatter 默认是 SimpleFormatter
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# 要自定义 SimpleFormatter 输出格式，可用以下范例：
#     <level>: <log message> [<date/time>]
# 例如：
# java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n
############################################################
# 特定名称空间 Logger 组态
############################################################
# 例如设定 com.xyz.foo 名称空间 Logger 的层级为 SEVERE
com.xyz.foo.level = SEVERE
```

修改这个.properties 后另存至程序 CLASSPATH 中，然后启动 JVM 时，指定 java.util.logging.config.file 系统属性为 .properties 名称，如 -Djava.util.logging.config.file=logging.properties，程序中的 Logger 就会套用指定文档中的组态设定。

提示»» 在 JDK8 或更早版本中，则可以在 JRE 目录的 lib 目录中找到 logging.properties 文件。

15.2 国际化基础

应用程序根据不同地区用户，呈现不同语言、日期格式等称为本地化(Localization)。应用程序在设计时，可在不修改应用程序情况下，根据不同用户直接采用不同语言、日期格式等，这样的设计考虑称为国际化(internationalization)，简称 i18n(因为 i 到 n 之间有 18 个字母)。国际化是个很复杂的议题，本节将介绍 java.util.ResourceBundle 与 java.util.Locale 的使用，可作为日后探讨国际化议题的基础。

15.2.1 使用 ResourceBundle

在 Java 如何处理 i18n 之前，必须先了解 Java 中如何处理非西欧字符，这在 4.4.3 节中已经介绍过，建议先回顾一下 4.3.3 节的内容。

在程序中有很多字符串信息会被写死在程序中，如果想要改变某个字符串信息，必须修改程序代码然后重新编译。例如，简单显示“Hello!World!”的程序就是这样：

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello!World!");
    }
}
```

就这个程序来说，如果日后想要改变“Hello!World!”为“Hello!Java!”，就要修改程序代码中的文字信息并重新编译。

对于日后可能变动的文字信息，可以考虑将信息移至程序之外，方法是使用 ResourceBundle 来做信息绑定。首先要准备一个.properties 文档，如 messages.properties，而文档内容如下：

I18N messages.properties

```
cc.openhome.welcome=Hello
cc.openhome.name=World
```

.properties 文档必须放置在 CLASSPATH 的路径设定下，文档中撰写的是键/值配对，之后在程序中可以使用键来获取对应的值。例如：

I18N Hello.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.ResourceBundle;

public class Hello {
    public static void main(String[] args) {
        ResourceBundle res = ResourceBundle.getBundle("messages");
        out.print(res.getString("cc.openhome.welcome") + "!");
        out.println(res.getString("cc.openhome.name") + "!");
    }
}
```

ResourceBundle 的静态 getBundle()方法会获取一个 ResourceBundle 的实例，所给定的自变量名称是信息文档的主文件名，getBundle()会自动找到对应的.properties 文档，获取 ResourceBundle 实例后，可以使用 getString()指定键来获取文档中对应的值，如果日后想要改变显示的信息，只要改变.properties 文档的内容就可以了。范例执行结果如下：

```
Hello!World!
```

15.2.2 使用 Locale

国际化的三个重要概念是地区(Locale)信息、资源包(Resource bundle)与基础名称(Base name)。

地区信息代表了特定的地理、政治或文化区，地区信息可由一个语言编码(Language code)与可选的地区编码(Country code)来指定。其中语言编码是 ISO-639(http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt)定义，由两个小写字母代表，如 ca 表示加拿大文(Catalan)，zh 表示中文(Chinese)。地区编码则由两个大写字母表示，定义在 ISO-3166(http://www. chemie.fu-berlin.de/diverse/doc/ISO_3166.html)，如 IT 表示意大利(Italy)，TW 表示中国台湾(Taiwan)。

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

地区信息的对应类是 Locale，在建立 Locale 实例时，可以指定语言编码与地区编码。例如，建立代表台湾繁体中文的 Locale，可以这样：

```
Locale locale = new Locale("zh", "TW");
```

资源包中包括了特定地区的相关信息，前面所介绍的 ResourceBundle 对象，就是 JVM 中资源包的代表对象。代表同一组信息但不同地区的各个资源包会共享相同的基础名称，使用 ResourceBundle 的 getBundle()时指定的名称，就是在指定基础名称。

例如，ResourceBundle 的 getBundle()若仅指定 messages，会尝试用默认 Locale(由 Locale.getDefault()获取的对象)获取.properties 文档。例如，若默认 Locale 代表 zh_TW，则 ResourceBundle 的 getBundle()若指定 messages，则会尝试获取 messages_zh_TW.properties 文档中的信息，若找不到，再尝试找 messages.properties 文档中的信息。

提示>>> 在 13.2 节曾经介绍过 DateFormat 与 Calendar，DateFormat 在使用静态方法 getDateInstance()、getTimeInstance()、getDateTimeInstance()时，或 Calendar 在使用静态方法 getInstance()时，若不指定 Locale，也会尝试用默认 Locale。

从 JDK9 开始支持 UTF-8 编码的.properties 文件，如果希望建立一个 messages_zh_TW.properties，并在当中建立台湾繁体中文的信息，从 JDK9 开始，只要 messages_zh_TW.properties 使用 UTF-8 编码就可以了。

若是以 JDK8 或更早版本，建立一个 messages_zh_TW.properties，并在其中建立台湾繁体中文的信息，并不是直接在 messages_zh_TW.properties 中撰写中文，而是必须使用 Unicode 编码表示(4.4.3 讲 Java 中字符串处理时提过)，通过 JDK 工具程序 native2ascii 来协助转换。例如，可以在 messages_zh_TW.txt 中撰写以下内容：

```
cc.openhome.welcome=哈啰
cc.openhome.name=世界
```

如果编辑器使用 MS950 编码，那么在 JDK8 或更早版本中，可以这样执行 native2ascii 程序：

```
> native2ascii -encoding MS950 messages_zh_TW.txt messages_zh_TW.properties
```

这样就会产生 messages_zh_TW.properties 文档，内容如下：

I18N messages_zh_TW.properties

```
cc.openhome.welcome=\u54c8\u56c9
cc.openhome.name=\u4e16\u754c
```

也就是 native2ascii 程序会将非 ASCII 字符转换为 Unicode 编码表示，如果想将 Unicode 编码表示的.properties 转回中文，则可以使用-reverse 自变量。例如，将上面的程序转回中文，并使用 UTF-8 编码文档储存：

```
> native2ascii -reverse -encoding UTF-8 messages_zh_TW.properties messages_zh_TW.txt
```

提示>>> 由 JDK9 支持 UTF-8 编码的.properties 文件，native2ascii 工具程序也就从 JDK9 中删除了。

如果执行前面的 Hello 类，而系统默认 Locale 为 zh_TW，则会显示“哈啰!世界!”的结果。如果提供 messages_en_US.properties：

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

I18N messages_en_US.properties

```
cc.openhome.welcome=Hi
cc.openhome.name=Earth
```

ResourceBundle 的 getBundle() 可以指定 Locale 对象，如果这样撰写程序：

```
Locale locale = new Locale("en", "US");
ResourceBundle res = ResourceBundle.getBundle("messages", locale);
System.out.print(res.getString("cc.openhome.welcome") + "!");
System.out.println(res.getString("cc.openhome.name") + "!");
```

则 ResourceBundle 会尝试获取 messages_en_US.properties 中的信息，结果就是显示“Hi!Earth!”。

总结一下使用 ResourceBundle 时，如何根据基础名称获取对应的信息文档：

(1) 使用指定的 Locale 对象获取信息文档。

(2) 使用 Locale.getDefault() 获取的对象获取信息文档。

(3) 使用基础名称获取信息文档。

提示 »» 在 NetBeans 中，右击 properties，在弹出的快捷菜单中选择“打开”命令，会出现属性编辑窗口，如果输入非西欧字符，而使用的 JDK 为 8 或更早版本，会自动转为 Unicode 编码储存至对应的.properties 文档，如图 15.3 所示。

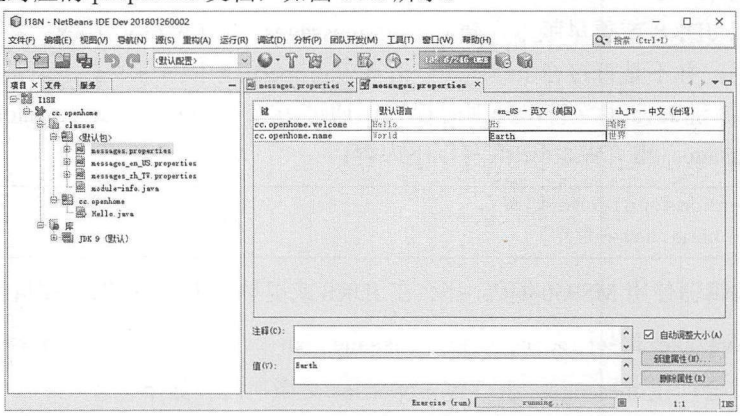


图 15.3 查看字符编码

15.3 规则表示式

规则表示式(Regular Expression)最早是由数学家 Stephen Kleene 于 1956 年提出，主要用于字符、字符串格式比较，后来在信息领域广为应用。Java 提供一些支持规则表示式操作的标准 API，下面将从如何定义规则表示式开始介绍。

15.3.1 规则表示式简介

如果想根据某个字符或字符串切割字符串，可以使用 String 的 split() 方法，它会返回切割后各子字符串组成的 String 数组。例如：

严禁网络传播本PDF，违者责任自负！

RegularExpression SplitDemo.java

```
package cc.openhome;

public class SplitDemo {
    public static void main(String[] args) {
        // 根据逗号切割
        for(String token : "Justin,Monica,Irene".split(",")) {
            System.out.println(token);
        }
        // 根据 Orz 切割
        for(String token : "JustinOrzMonicaOrzIrene".split("Orz")) {
            System.out.println(token);
        }
        // 根据 Tab 字符切割
        for(String token : "Justin\tMonica\tIrene".split("\\t")) {
            System.out.println(token);
        }
    }
}
```

执行结果会分别针对逗号、Orz、Tab字符对字符串进行切割：

```
Justin
Monica
Irene
Justin
Monica
Irene
Justin
Monica
Irene
```

实际上 String 的 split()方法接受的是规则表示式，范例中指定的方式是最简单的规则表示式：按照字面意义比较。最后一个 split()比较奇怪，为何是指定"\\t"？规则表示式是规则表示式，在 Java 中要将规则表示式撰写在""中是另一回事，所以首先得了解规则表示式如何定义。

规则表示式基本上包括两种字符：字面意义字符(Literals)与元字符(Metacharacters)。字面意义字符是指按照字面意义比较的字符，例如刚才在范例中指定的 Orz，指的是三个字面字符 O、r、z 的规则；元字符是不按照字面比较，在不同情境有不同意义的字符，例如^是元字符，规则表示式 ^Orz 是指行首立即出现 Orz 的情况，也就是此时 ^ 表示一行的开头，但规则表示式[^Orz]是指不包括 O 或 r 或 z 的比较，也就是在[]中时^表示非之后几个字符的情况。元字符就像是程序语言中的控制结构之类的语法，找出并理解元字符想要解释的概念，对于规则表示式的阅读非常重要。

1. 字面意义字符

字母和数字在规则表示式中，都是按照字面意义比较，有些字符之前加上了\之后，会被当作元字符。例如，\t 代表单击 Tab 键的字符。表 15.1 列出了规则表示式支持的字面意义字符。

严禁网络传播本PDF，违者责任自负！

表 15.1 字面意义字符

字 符	说 明
字母或数字	比较字母或数字
\\	比较\
\0n	八进制 0n 字符(0 ≤ n ≤ 7)
\0nn	八进制 0nn 字符(0 ≤ n ≤ 7)
\0mnn	八进制 0mnn 字符(0 ≤ m ≤ 3，0 ≤ n ≤ 7)
\xhh	十六进制 0xhh 字符
\uhhhh	十六进制 0xhhh 字符
\x{h...h}	十六进制 0xh...h 字符
\t	Tab(\u0009)
\n	换行(\u000A)
\r	返回(\u000D)
\f	换页(\u000C)
\a	响铃(\u0007)
\e	Esc(\u001B)
\cx	控制字符 x

元字符在规则表示式中有特殊意义，如! $ ^ * ( ) + = { } [ ] | \ : . ?等。若要比较这些字符，则必须加上忽略符号，例如要比较!，则必须使用\!，要比较$字符，则必须使用\$。如果不确定哪些标点符号字符要加上忽略符号，可以在每个标点符号前加上\，例如比较逗号也可以写\,。

如果规则表示式为 XY，那么表示比较“X 之后要跟随着 Y”；如果想表示“X 或 Y”，可以使用 X|Y；如果有多个字符要用“或”的方式表示，例如“X 或 Y 或 Z”，则可以使用稍后会介绍的字符类表示为[XYZ]。

使用 Java 字符串撰写规则表示式比较麻烦。若有个 Java 字符串是"Justin+Monica+Irene"，想使用 split()方法依+切割，要使用的规则表示式是\+，要将\+放至""之间时，按照 Java 字符串的规定，必忽略\+的\，所以必须撰写为"\\+"。类似地，如果有个 Java 字符串是"Justin||Monica||Irene"，你想用 split()方法依||切割，要使用的规则表示式是\|\|，要将\|\|放至""之间时，按照 Java 字符串规定必须忽略\|的\，就必须撰写为"\\|\\|"。例如：

```
// 规则表示式\|\|撰写为 Java 字符串是"\\|\\|"
for(String token : "Justin||Monica||Irene".split("\\|\\|")) {
    out.println(token);
}
```

如果有个字符串是"Justin\\Monica\\Irene"，也就是原始文字是 Justin\Monica\Irene 以 Java 字符串表示，若想使用 split()方法依\切割，要使用的规则表示式是\\，那就得这样撰写：

```
// 规则表示式\\撰写为 Java 字符串是"\\\\"
for(String token : "Justin\\Monica\\Irene".split("\\\\")) {
    out.println(token);
}
```

注意，规则表示式是规则表示式，在 Java 中要将规则表示式撰写在""中是另一回事，不要将两个混为一谈。

提示 »» 在 Java 中使用字符串撰写规则表示式时，先写下规则表示式，再在每个\前加上\。在 NetBeans 中如果在""中贴入文字，会自动在原文字的\前加上\。

严禁网络传播本PDF，违者责任自负！

2. 字符类

规则表示式中，多个字符可以分归在一起，成为一个字符类(Character class)，字符类会比较文字中是否有“任一个”字符符合字符类中某个字符。归类字符的方式之一是将字符放于[]中。例如，若文字为Justin1Monica2Irene3Bush，你想要依1或2或3切割字符串，则规则表示式撰写为[123]：

```
for(String token : "Justin1Monica2Irene3".split("[123]")) {
    out.println(token);
}
```

规则表示式123连续出现字符1、2、3，然而[]中的字符是“或”的意思，也就是[123]表示“1或2或3”，|在字符类中只是个字面意义字符，不会被当作“或”来表示。字符类中可以使用连字符-作为字符类元字符，表示一段文字范围，例如要比较文字中是否有1到5任一数字出现，规则表示式为[1-5]；要比较文字中是否有a到z任一字母出现，规则表示式为[a-z]；要比较文字中是否有1到5、a到z、M到W任一字符出现，规则表示式可以写为[1-5a-zM-W]。字符类中可以使用^作为字符类元字符，[^]则为反字符类(Negated character class)，例如[^abc]会比较a、b、c以外的字符。表15.2所示为字符类范例。

表15.2 字符类

字符类	说明
[abc]	a或b或c任一字符
[^abc]	a、b、c以外的任一字符
[a-zA-Z]	a到z或A～Z任一字符
[a-d[m-p]]	a到d或m～p任一字符(并集)，等于[a-dm-p]
[a-z&&[def]]	a到z而且是d、e、f的任一字符(交集)，等于[def]
[a-z&&[^bc]]	a到z而且不是b或c的任一字符(减集)，等于[ad-z]
[a-z&&[^m-p]]	a到z而且不是m到p的任一字符，等于[a-lq-z]

可以看到，字符类中可以再有字符类，把规则表示式想成是语言的话，字符类就像是其中独立的子语言。

有些字符类很常用，例如经常会比较是否为0到9的数字，可以撰写为[0-9]，或是撰写为\d，这称为预定义字符类(Predefined character class)，它们不用被包括在[]之中。表15.3列出了可用的预定义字符类。

表15.3 预定义字符类

预定义字符类	说明
.	任一字符
\d	比较任一数字字符，即[0-9]
\D	比较任一非数字字符，即[^0-9]
\s	比较任一空格符，即[\t\n\x0B\f\r]
\S	比较任一非空格符，即[^\s]
\w	比较任一ASCII字符，即[a-zA-Z0-9_]
\W	比较任一非ASCII字符，即[^\w]

严禁网络传播本PDF，违者责任自负！

java.util.regex.Pattern 文件说明中，还列出了一些可用的字符类，建议必要时自行参考 API 文件。

3. 贪婪、逐步、独吐量词

如果想让用户输入的手机号码格式为 xxxx-xxxxxx，其中 x 为数字，虽然规则表示式可以使用\d\d\d\d-\d\d\d\d\d\d，不过更简单的写法是\d{4}-\d{6}。{n}是贪婪量词(Greedy quantifier)表示法的一种，表示前面的项目出现 n 次。表 15.4 列出了可用的贪婪量词。

表 15.4 贪婪量词

贪婪量词	说明
X?	X 项目出现一次或没有
X*	X 项目出现零次或多次
X+	X 项目出现一次或多次
X{n}	X 项目出现 n 次
X{n,}	X 项目至少出现 n 次
X{n,m}	X 项目出现 n 次但不超过 m 次

贪婪量词之所以贪婪，是因为看到贪婪量词时，比较器(Matcher)会把剩余文字整个吃掉，再逐步吐出(Back-off)文字，看看是否符合贪婪量词后的规则表示式。如果吐出部分符合，而吃下部分也符合贪婪量词就比较成功，结果就是贪婪量词会尽可能地找出长度最长的符合文字。

例如文字 xfooxxxxxxfoo，若使用规则表示式.*foo 比较，比较器会先吃掉整个 xfooxxxxxxfoo，再吐出 foo 符合 foo 部分，剩下的 xfooxxxxxx 也符合.*部分，所以得到的符合字符串就是整个 xfooxxxxxxfoo。

如果在贪婪量词表示法后加上?，将会成为逐步量词(Reluctant quantifier)，又常称为懒惰量词，或非贪婪(non-greedy)量词(相对于贪婪量词来说)。比较器看到逐步量词时，会一边吃掉剩余文字，一边看看吃下的文字是否符合规则表示式，结果就是逐步量词会尽可能地找出长度最短的符合文字。

例如文字 xfooxxxxxxfoo，若用规则表示式.*?foo 比较，比较器在吃掉 xfoo 后发现符合*?foo，接着继续吃掉 xxxxxxfoo 发现符合，所以得到 xfoo 与 xxxxxxfoo 两个符合文字。

如果在贪婪量词表示法后加上+，将会成为独吐量词(Possessive quantifier)，比较器看到独吐量词时，会先将剩余文字吃掉，然后看看独吐量词部分是否符合吃下的文字，如果符合就不会再吐出来了。

例如文字 xfooxxxxxxfoo，若使用规则表示式.*+foo 比较，比较器会先吃掉整个 xfooxxxxxxfoo，结果.*+就可以符合 xfooxxxxxxfoo 了，所以比较器就不会再吐出文字，因为没有剩余文字符合 foo 部分，所以结果就是没有任何文字符合。

表 15.5 整理了前面三个量词的讨论。

表 15.5 比较 xfooxxxxxxfoo

规则表示式	得到的符合文字
.*foo	xfooxxxxxxfoo
.*?foo	xfoo 与 xxxxxxfoo
.*+foo	无

严禁网络传播本PDF，违者责任自负！

下面这个范例使用 String 的 replaceAll()来示范三个量词的差别。

RegularExpression ReplaceDemo.java

```
package cc.openhome;

public class ReplaceDemo {
    public static void main(String[] args) {
        String[] regexs = {".*foo", ".*?foo", ".*+foo"};
        for(String regex : regexs) {
            System.out.println("xfooxxxxxxfoo".replaceAll(regex, "Orz"));
        }
    }
}
```

replaceAll()会将符合规则表示式的字符串取代后返回新字符串，出现几次 Orz，就可以知道符合的字符串有几个：

```
Orz
OrzOrz
xfooxxxxxxfoo
```

提示»» 读者可以参考 Quantifiers，了解更多有关量词的使用，网址如下：
http://docs.oracle.com/javase/tutorial/essential/regex/quant.html

4. 边界比较

如果有个文字 Justin dog Monica doggie Irene，你想要以当中单词 dog 切出前后两个子字符串，也就是 Justin 与 Monica doggie Irene 两个部分，那么下面程序会让你失望：

RegularExpression SplitDemo2.java

```
package cc.openhome;

public class SplitDemo2 {
    public static void main(String[] args) {
        for(String str : "Justin dog Monica doggie Irene".split("dog")) {
            System.out.println(str.trim());
        }
    }
}
```

在这个范例程序中，doggie 因为当中有 dog 子字符串，也被当作切割的依据，所以执行结果会是：

```
Justin
Monica
gie Irene
```

可以使用\b 标出单词边界，如\bdog\b，这就只会比较出 dog 单词。例如：

RegularExpression SplitDemo3.java

```
package cc.openhome;
```


严禁网络传播本PDF，违者责任自负！

```
public class SplitDemo3 {
    public static void main(String[] args) {
      for(String str : "Justin dog Monica doggie Irene".split("\\bdog\\b")) {
            System.out.println(str.trim());
        }
    }
}
```

执行结果如下：

```
Justin
Monica doggie Irene
```

边界比较用来表示文字必须符合指定的边界条件，也就是定位点，因此这类表示式也常称为锚点(Anchor)，表 15.6 列出了规则表示式中可用的边界比较。

表 15.6 边界比较

边界比较	说 明
^	一行开头
$	一行结尾
\b	单词边界
\B	非单词边界
\A	输入开头
\G	前一个符合项目结尾
\Z	非最后终端机(final terminator)的输入结尾
\z	输入结尾

5. 分组与参考

可以使用()来将规则表示式分组，除了作为子规则表示式之外，还可以搭配量词使用。例如想要验证电子邮件格式，允许的用户名称开头要是大小写英文字符，之后可搭配数字，规则表示式可以写为^[a-zA-Z]+\d*；因为@后域名可以有数层，必须是大小写英文字符或数字，规则表示式可以写为([a-zA-Z0-9]+\.)+，其中使用()群组了规则表示式，之后的+表示这个群组的表示式符合一次或多次，最后要是 com 结尾，整个结合起来的规则表示式就是^[a-zA-Z]+\d*@([a-zA-Z0-9]+\.)+com。

被分组的规则表示式，还可以在稍后回头参考(Back reference)。在这之前，必须知道分组计数，如果有个规则表示式((A)(B(C)))，其中有 4 个分组，这是以遇到的左括号来计数，所以 4 个分组分别是：

(1) ((A)(B(C)))

(2) (A)

(3) (B(C))

(4) (C)

分组回头参考时，是在\后加上分组计数，表示参考第几个分组的比较结果。例如，\d\d 要求比较两个数字，(\d\d)\1 的话，表示要输入 4 个数字，输入的前两个数字与后两个数字必须相同，例如输入 1212 会符合，因为 12 符合(\d\d)，\1 要求接下来输入也要是 12；若输入 1234

严禁网络传播本PDF，违者责任自负！

则不符合，因为12符合(\d\d)，\1要求接下来的输入也要是12，然而接下来的数字是34，因而不符合。

再来看个实用的例子，["'][^"']* ["']比较单引号或双引号中0或多个字符，但没有比较两个都要是单引号或双引号，(["'])[^"']*\1则比较出前后引号必须一致。

提示 »» 想要得到有关规则表示式更完整的说明，除了可以参考 **java.util.regex.Pattern** 文件说明，也可参考以下网址(*Lesson: Regular Expressions*)：
http://docs.oracle.com/javase/tutorial/essential/regex/

15.3.2 Pattern 与 Matcher

在程序中使用规则表示式，必须先针对规则表示式做剖析、验证等动作，确定规则表示式语法无误，对字符串进行比较。在频繁使用某规则表示式的场合，若可以将剖析、验证过后的规则表示式重复使用，对效率将会有所帮助。

java.util.regex.Pattern 实例是规则表示式在JVM中的代表对象，Pattern 的构造函数被标示为 private，所以无法用 new 创建 Pattern 实例，而必须通过 Pattern 的静态方法 compile() 来获取。compile()方法在剖析、验证过规则表示式无误后，将会返回 Pattern 实例，之后就可以重复使用这个实例。例如：

```
Pattern pattern = Pattern.compile(".*foo");
```

Pattern.compile()方法的另一版本，可以指定旗标(Flag)。例如，想不分大小写比较 dog 文字，可以这样：

```
Pattern pattern = Pattern.compile("dog", Pattern.CASE_INSENSITIVE);
```

也可以在规则表示式中使用嵌入旗标表示法(Embedded Flag Expression)。例如，Pattern.CASE_INSENSITIVE 等效的嵌入旗标表示法为(?i)，以下片段效果等同上例：

```
Pattern pattern = Pattern.compile("(?i)dog");
```

规则表示式本身可读性差、除错不易，如果因规则表示式有误而导致 compile()调用失败，会抛出 java.util.regex.PatternSyntaxException，可以使用 getDescription()获取错误说明，使用 getIndex()获取错误索引，使用 getPattern()获取错误的规则表示式，getMessage()会以多行显示错误的索引、描述等综合信息。

在获取 Pattern 实例后，可以使用 split()方法将指定字符串以规则表示式切割，效果等同于使用 String 的 split()方法；可以使用 matcher()方法指定要比较的字符串，这会返回 java.util.regex.Matcher 实例，表示对指定字符串的比较器；可以使用 find()方法看看是不是有下一个符合字符串，或是使用 lookingAt()看看字符串开头是否符合规则表示式，使用 group()方法则可以返回符合的字符串。例如：

RegularExpression PatternMatcherDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.regex.*;

public class PatternMatcherDemo {
```

```
    public static void main(String[] args) {
        String[] regexs = {".*foo", ".*?foo", ".*+foo"};
        for(String regex : regexs) {
            Pattern pattern = Pattern.compile(regex);
            Matcher matcher = pattern.matcher("xfooxxxxxxfoo");
            out.printf("%s find ", pattern.pattern());
            while(matcher.find()) {
                out.printf(" \"%s\"", matcher.group());
            }
            out.println(" in \"xfooxxxxxxfoo\".");
        }
    }
}
```

这个范例示范了贪婪、逐步与独吐量词的比对结果，执行结果如下：

```
.*foo find  "xfooxxxxxxfoo" in "xfooxxxxxxfoo".
.*?foo find  "xfoo" "xxxxxxfoo" in "xfooxxxxxxfoo".
.*+foo find  in "xfooxxxxxxfoo".
```

如果规则表示式中有分组，group()可以接受 int 整数指定分数计数。举例来说，规则表示式如果是((A)(B(C)))，若指定文字为 ABC，matcher.find()后指定 group(1)就是"ABC"，group(2)就是"A"，group(3)就是"BC"，group(4)就是"C"，由于分组计数会从 1 开始，因此 group(0)就相当于直接调用没有参数的 group()。

Matcher 还有 replaceAll()方法，可以将符合规则表示式的部分以指定的字符串取代，效果等同于 String 的 replaceAll()方法，replaceFirst()与 replaceEnd()则可分别取代首个、最后符合规则表示式的部分；start()方法可以获取符合字符串的起始索引，end()方法可获取符合字符串最后一个字符后的索引。

如果规则表示中有分组设定，在使用 replaceAll()时，可以使用$n 来捕捉被分组匹配的文字。例如，以下的片段可以将用户邮件地址从.com 取代为 .cc：

```
Pattern pattern = Pattern.compile("(^[a-zA-Z]+\\d*)@([a-z]+?.)com");
Matcher matcher = pattern.matcher("caterpillar@openhome.com");
out.println(matcher.replaceAll("$1@$2cc")); // caterpillar@openhome.cc
```

整个规则表示式匹配了"caterpillar@openhome.com"，第一个分组捕捉到"caterpillar"，第二个分组捕捉到"openhome。"，$1 与$2 就分别代表这两个部分。

下面这个范例可以让你输入规则表示式与想比较的字符串，执行结果将显示比较的结果：

RegularExpression Regex.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.*;
import java.util.regex.*;

public class Regex {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        try {
            out.print("输入规则表示式: ");
```

```
            String regex = console.nextLine();
            out.print("输入要比较的文字: ");
            String text = console.nextLine();
            print(match(regex, text));
        } catch(PatternSyntaxException ex) {
                out.println("规则表示式有误");
                out.println(ex.getMessage());
        }
    }

    private static List<String> match(String regex, String text) {
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(text);
        List<String> matched = new ArrayList<>();
        while(matcher.find()) {
            matched.add(String.format(
                    "从索引 %d 开始到索引 %d 之间找到符合文字 \"%s\"%n",
                    matcher.start(), matcher.end(), matcher.group()));
        }
        return matched;
    }

     private static void print(List<String> matched) {
        if(matched.isEmpty()){
            out.println("找不到符合文字");
        }
        else {
            matched.forEach(out::println);
        }
    }
}
```

执行结果如下所示：

```
输入规则表示式: .*?foo
输入要比较的文字: xfooxxxxxxfoo
从索引 0 开始到索引 4 之间找到符合文字 "xfoo"
从索引 4 开始到索引 13 之间找到符合文字 "xxxxxxfoo"
```

在第 12 章中介绍过，JDK8 新增了 Stream API，在 Pattern 中也因 Stream API 而新增了 **splitAsStream()** 静态方法，它返回的是 Stream<String>，适用于需要管线化、惰性操作的场合。

15.4 处理数字

在 4.1.2 中曾经简单介绍过 BigDecimal 的使用，如果需要处理大数或者是精度等更复杂的数字问题，就必须对 BigInteger 或 BigDecimal 有更多的认识，而这一节也会介绍到如何对数字进行格式化。

15.4.1 使用 BigInteger

在 Java 中，基本数据类型的整数最大值是 9223372036854775807L，最小值是-9223372036854775808L，

若想要表示的整数超过范围，可以使用 java.math.BigInteger。例如，想要表示 9223372036854775808 这个数字，可以如下建立 BigInteger 实例：

```
jshell> BigInteger n = new BigInteger("9223372036854775808");
n ==> 9223372036854775808

jshell> n.add(BigInteger.ONE);
$2 ==> 9223372036854775809

jshell> n.add(BigInteger.ONE).add(BigInteger.TWO).add(BigInteger.TEN);
$3 ==> 9223372036854775821
```

由于 9223372036854775808 这个数字超过了 long 可储存的范围，创建 BigInteger 时不能直接写 9223372036854775808L，表示该数字的一个方式是使用字符串。而在建立 BigInteger 实例之后，若要进行加、减、乘、除等运算，可以使用 add()、subtract()、multiply()、divide() 等方法，这些方法接受的自变量类型也是 BigInteger，每个方法调用过后会返回新的 BigInteger 实例，因此可以形成流畅的调用风格。

由于在整数运算中，0、1、2、10 之类的数字很常使用，BigInteger 本身提供了 BigInteger.ZERO、BigInteger.ONE、BigInteger.TWO、BigInteger.TEN 实例，这样的话就不用自行创建这些数字的 BigInteger 实例。

除了基本的算术运算之外，在余除运算上，BigInteger 提供了 mod() 与 divideAndRemainder() 方法，后者会返回长度为 2 的 BigInteger 数组，索引 0 为商数、索引 1 为余数：

```
jshell> BigInteger.TEN.mod(BigInteger.TWO);
$1 ==> 0

jshell> BigInteger.TEN.divideAndRemainder(BigInteger.TWO);
$2 ==> BigInteger[2] { 5, 0 }
```

在比较、条件运算上，BigInteger 提供了 equals() 与 compareTo() 方法，位运算上提供了 and()、or()、xor()、not()，对于左移(<<)与右移(>>)两个运算符，则提供了 shiftLeft()、shiftRight() 方法。

```
jshell> BigInteger.ONE.shiftLeft(1);
$1 ==> 2

jshell> BigInteger.ONE.shiftLeft(1).shiftLeft(2);
$2 ==> 8
```

如果必须在基本类型整数与 BigInteger 之间转换，BigInteger.valueOf() 接受 long 类型整数转换为 BigInteger。提供 valueOf() 方法而不是提供接受 long 的构造函数的原因在于，valueOf() 方法会维护或重用已创建之 BigInteger 实例。

若想将 BigInteger 转换为基本类型，可以使用 intValue()、longValue() 方法，然而若原本的数字无法在 int 或 long 中容纳，这两个方法返回的整数会失去精度，若想在原本的数字无法在 short、int、long 中容纳时抛出 ArithmeticException 异常，可以使用 shortValueExtract()、intValueExtract()、longValueExtract() 方法。

若有必要，也可以使用 floatValue()、doubleValue() 方法，将 BigInteger 代表的数字转换为基本类型的 float 或 double。

15.4.2 使用 BigDecimal

正如4.1.2中看过的例子，如果你在意浮点数误差的问题，应该使用 BigDecimal，而不是直接使用 float、double 基本类型来表示浮点数。而创建 BigDecimal 实例时，最基本的方式就是使用字符串来表示 BigDecimal 实例将代表的浮点数。

在建立 BigDecimal 实例之后，若要对其进行运算，方式与 BigInteger 类似，基本上都有对应的方法可以操作，不过查看 API 文件，会发现这些方法上有的必须指定量级(scale)或进位舍去模式(java.math.RoundingMode)。

理解量级最简单的方式，就是查看 API 文件中 BigDecimal 其中一个构造函数说明，如图 15.4 所示。

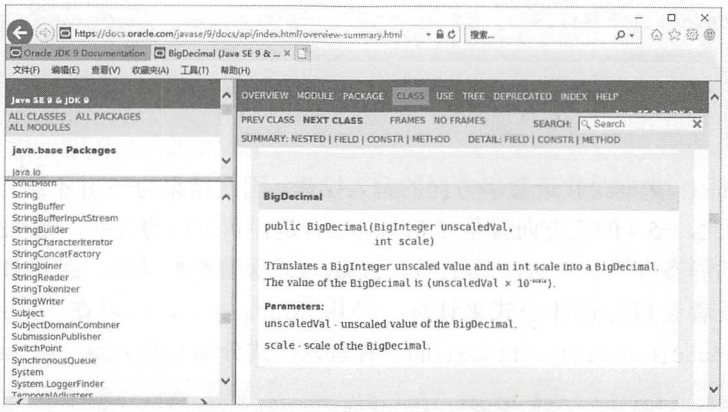


图 15.4 BigDecimal 构造函数之一

因此，若有个非量级值(unscaledVal)为 360，而量级为 2，那么就表示 3.60 这个浮点数，若量级为 5，就表示 0.00360。量级可以是负数，若量级为-2，表示 360 乘上 100。

```
jshell> new BigDecimal(BigInteger.valueOf(360), 2);
$1 ==> 3.60

jshell> new BigDecimal(BigInteger.valueOf(360), 5);
$2 ==> 0.00360

jshell> new BigDecimal(BigInteger.valueOf(360), -2);
$3 ==> 3.60E+4
```

在建立 BigDecimal 实例之后，可以使用 unscaledValue()获取非量级值，使用 scale()方法获取量级。若要设定量级，可以使用 setScale()方法，这会返回新的 BigDecimal 实例。由于量级改变可能会导致位数的减少，因此必须知道该怎么处理进位舍去，这时就必须指定 RoundingMode。

提示>>> BigDecimal 上使用 int 列举的进位舍去常数，在 JDK9 之后被标示为废弃了，相关的方法也同样标示为废弃。

许多人最熟悉的进位舍入法应该是四舍五入、无条件进位或舍去，这里先使用正值会比较容易理解，5.4 套用这三种舍入法取整数的结果，分别是：5、6.0 与 5.0。而在 Java 中，Math 的静态方法 round()、ceil()与 floor()可用来做对应的计算：

```
jshell> Math.round(5.4);
$1 ==> 5

jshell> Math.ceil(5.4);
$2 ==> 6.0

jshell> Math.floor(5.4);
$3 ==> 5.0
```

那么，-5.4呢？套用这三个方法的结果，分别会是：-5、-5.0、-6.0。

```
jshell> Math.round(-5.4);
$4 ==> -5

jshell> Math.ceil(-5.4);
$5 ==> -5.0

jshell> Math.floor(-5.4);
$6 ==> -6.0
```

实际上，round()是向最接近数字方向的舍入操作，因此结果为-5并不意外；ceil()是往正方向的舍入，因此，-5.4的正方向就是-5.0；floor()是往负方向的舍入，-5.4的负方向是-6.0。

由于Math的静态方法round()、ceil()、floor()只保留整数部分，遇到想指定小数字数的时候，许多开发者会自行设计公式来计算。使用BigDecimal，可以在一些计算操作时指定RoundingMode，round()、ceil()、floor()的三种舍入模式分别对应HALF_UP、CEILING与FLOOR。

```
jshell> BigDecimal n = new BigDecimal(BigInteger.valueOf(54), 1);
n ==> 5.4

jshell> n.setScale(0, RoundingMode.HALF_UP);
$2 ==> 5

jshell> n.setScale(0, RoundingMode.CEILING);
$3 ==> 6

jshell> n.setScale(0, RoundingMode.FLOOR);
$4 ==> 5

jshell> BigDecimal n2 = new BigDecimal(BigInteger.valueOf(-54), 1);
n2 ==> -5.4

jshell> n2.setScale(0, RoundingMode.HALF_UP);
$6 ==> -5

jshell> n2.setScale(0, RoundingMode.CEILING);
$7 ==> -5

jshell> n2.setScale(0, RoundingMode.FLOOR);
$8 ==> -6
```

除了HALF_UP、CEILING与FLOOR之外，RoundingMode的UP模式，会远离0的方向，被舍弃的部分若不是0，左边的数字一律递增1，因此在UP模式下，5.4若只保留整数，操作后会是6，-5.4操作后，会是-6。RoundingMode的DOWN模式会接近0的方向，5.4操作后，会是5，-5.4操作后，会是-5。

HALF_UP 以及另一个 HALF_DOWN 模式，都会向最接近数字方向，进行舍入操作，不同的是，若最接近的数字距离相同，前者进位而后者舍去，因此，5.5 的 HALF_UP 会是 6，而 HALF_DOWN 会是 5。

HALF_EVEN 模式比较难理解一些，虽然会往最接近数字方向进行舍入操作，不过，若最接近的数字距离相同，是向相邻的偶数舍入，因此 HALF_EVEN 时，若舍去部分的左边为奇数，那么行为就如 HALF_UP，因此 5.5 会成为 6；若为偶数，那么就如 HALF_DOWN，因此 4.5 会成为 4。而这种舍入法又称银行家舍入法(Banker's rounding)，或者四舍五入取偶数(round-to-even)。

提示 »» 读者必须注意的是，搞清楚使用的语言或链接库对于舍入的行为究竟为何，Java 的 Math.round()采用的是 HALF_UP。不过，其他语言或链接库不见得是如此，例如 Python 3 的 round()函式采用银行家舍入法！因此，Python 3 中 round(5.5)是 6，而 round(4.5)会是 4。

在创建 BigDecimal 时，有的构造函数可以指定 java.math.MathContext 实例，而有的方法也接受 MathContext。MathContext 本身也提供了 DECIMAL128、DECIMAL64、DECIMAL32 常数来表示对应的 IEEE 754R 浮点数格式。自行创建 MathContext 实例时，可以指定最大精度(Precision)，这是指从数字最左边不是 0 的数字开始，直到最右边使用的数字个数。

BigDecimal 本身也有个 precision()可以获取目前浮点数使用之精度，例如：

```
jshell> BigDecimal n = new BigDecimal(BigInteger.valueOf(42), 5);
n ==> 0.00042

jshell> n.precision();
$2 ==> 2

jshell> BigDecimal n2 = new BigDecimal(BigInteger.valueOf(142), 5);
n2 ==> 0.00142

jshell> n2.precision();
$4 ==> 3
```

提示 »» 因此，使用 BigDecimal 处理小数字数时，记得是使用 scale 相关参数，或者使用 scale 相关方法，而不是精度。

15.4.3 数字的格式化

如果想要对数字进行格式化，可以使用 java.text.NumberFormat，这是个抽象类，但它提供了 getInstance()、getIntegerInstance()、getCurrencyInstance()、getPercentInstance()等方法，它们返回的实例类型是 NumberFormat 的子类，各自提供了默认的数字格式。

```
jshell> import java.text.NumberFormat;

jshell> NumberFormat.getInstance().format(123456789.987654321)
$2 ==> "123,456,789.988"

jshell> NumberFormat.getIntegerInstance().format(123456789.987654321);
$3 ==> "123,456,790"

jshell> NumberFormat.getCurrencyInstance().format(123456789.987654321);
$4 ==> "$123,456,789.99"
```

```
jshell> NumberFormat.getPercentInstance().format(123456789.987654321);
$5 ==> "12,345,678,999%"
```

这些方法都有个接受 Locale 的版本，例如想要显示日币符号的格式：

```
jshell> import static java.util.Locale.JAPAN;

jshell> NumberFormat.getCurrencyInstance(JAPAN).format(123456789.987654321);
$6 ==> "￥123,456,790"
```

NumberFormat 实例有个 parse()方法，可以指定代表数字的字符串，将之剖析为 Number 实例，在可能的情况下会返回 Long，否则的话就使用 Double，它们都是 Number 的子类。

```
jshell> NumberFormat.getCurrencyInstance(JAPAN).parse("￥123,456,790");
$7 ==> 123456790
```

NumberFormat 的父类是 java.text.Format，通过从 Format 继承下来的 format()方法，基本上也可以对 BigDecimal 进行格式化：

```
jshell> NumberFormat.getInstance().format(new BigDecimal("123456789.98765"));
$8 ==> "123,456,789.988"
```

必要时，可以指定格式字符串来自行创建 DecimalFormat 实例，它是 NumberFormat 的子类。例如，想要整数部分每四位数一个逗号，小数最多三个位数，程序撰写如下：

```
jshell> import java.text.*;

jshell> NumberFormat formatter = new DecimalFormat("#,####.###");
formatter ==> java.text.DecimalFormat@674dc

jshell> formatter.format(123456789.987654321);
$3 ==> "1,2345,6789.988"

jshell> formatter.format(123456789.92);
$4 ==> "1,2345,6789.92"
```

如果想要整数部分每四位数一个逗号，小数三个位数，不足补 0，程序撰写如下：

```
jshell> import java.text.*;

jshell> NumberFormat formatter = new DecimalFormat("#,####.000");
formatter ==> java.text.DecimalFormat@674dc

jshell> formatter.format(123456789.92);
$3 ==> "1,2345,6789.920"
```

#、0 等字符设定都有其意义，可参考 DecimalFormat 的 API 说明，了解每个字符的设定意义。

提示 »» NumberFormat 的 getInstance()等静态方法，基本上是返回 DecimalFormat 实例，不过根据 API 文件的说明，在某些特别的国家区域设定下，有可能返回的不是 DecimalFormat 实例。

15.5 再谈堆栈追踪

在 8.1.5 中曾经介绍过堆栈追踪，除了在发生并捕捉例外时可进行堆栈追踪之外，必要时也可以自行建立 Throwable 或获取 Thread，通过 getStackTrace() 方法获取堆栈追踪，而在 JDK9 中新增了 Stack-Walking API，提供了更多的选项，让堆栈追踪更有效率。

15.5.1 获取 StackTraceElement

在 8.1.5 中曾经看过异常发生时，利用异常对象自动收集的 Stack Frame 来显示堆栈追踪，当时还没介绍到多线程，自动收集的 Stack Frame 是来自于主线程的 JVM Stack。实际上，每个线程都会有自己专属的 JVM Stack。JVM Stack 是个先进后出结构，每调用一个方法，JVM 就会建立一个 Stack Frame 来储存局部变量、方法、类等信息并置放至 JVM Stack，方法调用结束后 Stack Frame 就从 JVM Stack 中跳出销毁。

也许是为了了解应用程序的行为，或者是进行除错，自行获取堆栈追踪有其需求，为此，JDK 从 1.4 开始就在 **Throwable** 与 **Thread** 类上提供了 **getStackTrace()** 方法，用来获取当时活动线程的 JVM Stack，并以 **StackTraceElement** 数组返回 JVM Stack 中全部的 Stack Frame。

如果想主动获取 Stack Frame，方式之一是建立 Throwable 实例并调用其 getStackTrace() 方法：

```
StackTraceElement[] stackTrace  = new Throwable().getStackTrace();
```

另一个方式是获取 Thread 实例，例如通过 Thread.currentThread()，并调用其 getStackTrace() 方法：

```
StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();
```

每个 StackTraceElement 实例代表着 JVM Stack 中的一个 Stack Frame，可以使用 StackTraceElement 的 getClassName()、getFileName()、getLineNumber()、getMethodName()，或者是 JDK9 新增的 getModuleName()、getModuleVersion()、getClassLoaderName() 等方法获取对应的信息。例如，改写 8.1.5 的范例，主动显示堆栈追踪：

StackTrace\StackTraceDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.List;

public class StackTraceDemo {
    public static void main(String[] args) {
        c();
    }

    static void c() {
        b();
    }
```

```
    static void b() {
        a();
    }

    static void a() {
        Thread currentThread = Thread.currentThread();
        StackTraceElement[] stackTrace = currentThread.getStackTrace();

        out.printf("Stack trace of thread %s:%n", currentThread.getName());
        List.of(stackTrace).forEach(out::println);
    }
}
```

这个范例程序中，c()方法调用b()方法，b()方法调用a()方法，因而堆栈追踪显示结果如下：

```
Stack trace of thread main:
java.base/java.lang.Thread.getStackTrace(Thread.java:1654)
cc.openhome/cc.openhome.StackTraceDemo.a(StackTraceDemo.java:21)
cc.openhome/cc.openhome.StackTraceDemo.b(StackTraceDemo.java:16)
cc.openhome/cc.openhome.StackTraceDemo.c(StackTraceDemo.java:12)
cc.openhome/cc.openhome.StackTraceDemo.main(StackTraceDemo.java:8)
```

每个线程会有自己的JVM Stack，因此下面的程序会显示线程个别的追踪：

StackTrace StackTraceDemo2.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.List;

public class StackTraceDemo2 {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> c());
        t.start();
        t.join(); // 这是为了能循环显示个别的堆栈追踪

        c();
    }

    static void c() {
        b();
    }

    static void b() {
        a();
    }

    static void a() {
        Thread currentThread = Thread.currentThread();
        StackTraceElement[] stackTrace = currentThread.getStackTrace();

        out.printf("Stack trace of thread %s:%n", currentThread.getName());
```

```
        List.of(stackTrace).forEach(out::println);
    }
}
```

执行结果如下：

```
Stack trace of thread Thread-0:
java.base/java.lang.Thread.getStackTrace(Thread.java:1654)
cc.openhome/cc.openhome.StackTraceDemo2.a(StackTraceDemo2.java:24)
cc.openhome/cc.openhome.StackTraceDemo2.b(StackTraceDemo2.java:19)
cc.openhome/cc.openhome.StackTraceDemo2.c(StackTraceDemo2.java:15)
cc.openhome/cc.openhome.StackTraceDemo2.lambda$main$0(StackTraceDemo2.java:8)
java.base/java.lang.Thread.run(Thread.java:844)
Stack trace of thread main:
java.base/java.lang.Thread.getStackTrace(Thread.java:1654)
cc.openhome/cc.openhome.StackTraceDemo2.a(StackTraceDemo2.java:24)
cc.openhome/cc.openhome.StackTraceDemo2.b(StackTraceDemo2.java:19)
cc.openhome/cc.openhome.StackTraceDemo2.c(StackTraceDemo2.java:15)
cc.openhome/cc.openhome.StackTraceDemo2.main(StackTraceDemo2.java:11)
```

如果只是想获取简单的信息，基本上可以使用以上方式。然而，getStackTrace()会获取全部的 Stack Frame，如果你只是想查看前几个，就会显得没有效率。若是想获取方法所在的类别信息呢？StackTraceElement 只提供 getClassName()返回字符串，你得自己想办法，例如通过反射(Reflection)(第 17 章会介绍)等机制来达到目的。如果想要进一步获取调用方法相关的类信息呢？JDK8 或以前的版本，没有标准 API 可以从事这个任务。

提示»» 真的要获取调用方法相关的类信息的话，在过去版本的 JDK 中，内部 API 的 sun.reflect.Reflection 中有个 getCallerClass()方法。

15.5.2 JDK9 的 Stack-Walking API

为了在堆栈追踪时更为方便，JDK9 新增了 Stack-Walking API。具体来说，可以通过 java.lang.StackWalker 的 getInstance()方法，获取 StackWalker 实例后，运用 forEach()方法来循序走访 StackWalker.StackFrame，每个 StackFrame 代表着 JVM Stack 中的一个 Stack Frame。例如，可改写先前的 StackTraceDemo 如下：

StackTrace StackWalkerDemo.java

```
package cc.openhome;

import static java.lang.System.out;

public class StackWalkerDemo {
    public static void main(String[] args) {
        c();
    }

    static void c() {
        b();
    }
```

```
    static void b() {
        a();
    }

    static void a() {
        out.printf("Stack trace of thread %s:%n",
                   Thread.currentThread().getName());
        StackWalker stackWalker = StackWalker.getInstance();
        stackWalker.forEach(out::println);
    }
}
```

堆栈追踪显示结果如下：

```
Stack trace of thread main:
cc.openhome/cc.openhome.StackWalkerDemo.a(StackWalkerDemo.java:21)
cc.openhome/cc.openhome.StackWalkerDemo.b(StackWalkerDemo.java:15)
cc.openhome/cc.openhome.StackWalkerDemo.c(StackWalkerDemo.java:11)
cc.openhome/cc.openhome.StackWalkerDemo.main(StackWalkerDemo.java:7)
```

StackWalker 本身有个 getCallerClass()方法，而 StackFrame 有个 getDeclaringClass()，然而默认是不能从这两个方法获取 Class 实例，必须在调用 StackWalker 的 getInstance()时指定 StackWalker.Option 为 RETAIN_CLASS_REFERENCE，否则会引发 java.lang.UnsupportedOperationException。

例如，若只想显示类与方法名称，代码如下：

StackTrace StackWalkerDemo2.java

```
package cc.openhome;

import static java.lang.StackWalker.Option.RETAIN_CLASS_REFERENCE;
import static java.lang.System.out;

public class StackWalkerDemo2 {
    public static void main(String[] args) {
        c();
    }

    static void c() {
        b();
    }

    static void b() {
        a();
    }

    static void a() {
        out.printf("Stack trace of thread %s:%n",
                Thread.currentThread().getName());

        StackWalker stackWalker =
                StackWalker.getInstance(RETAIN_CLASS_REFERENCE);
        out.printf("Caller class %s%n",
                stackWalker.getCallerClass().getName());
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        stackWalker.forEach(stackFrame -> {
            out.printf("%s.%s%n",
                    stackFrame.getDeclaringClass(),
                    stackFrame.getMethodName());
        });
    }
}
```

堆栈追踪显示结果如下：

```
Stack trace of thread main:
Caller class cc.openhome.StackWalkerDemo2
class cc.openhome.StackWalkerDemo2.a
class cc.openhome.StackWalkerDemo2.b
class cc.openhome.StackWalkerDemo2.c
class cc.openhome.StackWalkerDemo2.main
```

如果你只对某几个 StackFrame 感兴趣，或者想对 StackFrame 做转换或过滤，可以使用 StackWalker 实例的 walk()方法，例如，只想找到第一个 StackFrame：

```
Optional<StackFrame> frame =stackWalker.walk(frameStream -> frameStream.findFirst());
```

在 walk()的 Lambda 表达式中，会传入 Stream<? super StackFrame>，而 Lambda 表达式的返回值就是 walk()的返回值。因此，可以惰性操作传入的 Stream<? super StackFrame>，基本上，Stream 定义的操作，像是 filter()、map()、collect()、count()等都可以使用。

有些 Stack Frame 与反射或者特定的 JVM 特定操作相关，在不指定参数的情况下，StackWalker.getInstance()默认是不可获取 Class 实例且不显示这些 Stack Frame，如果想要显示反射相关的 Stack Frame，可以使用 SHOW_REFLECT_FRAMES，若想显示隐藏的 Stack Frame(包含反射相关 Stack Frame)，可以使用 SHOW_HIDDEN_FRAMES。

下面是个简单的范例，使用了反射机制来调用 c()方法，暂且忽略反射机制程序代码如何撰写，将重点放在不同选项下，StackWalker 访问的 Stack Frame 有何不同。

StackTrace StackWalkerDemo3.java

```
package cc.openhome;

import static java.lang.StackWalker.Option.*;
import static java.lang.System.out;
import java.util.List;

public class StackWalkerDemo3 {
    public static void main(String[] args) throws Exception {
        StackWalkerDemo3.class.getDeclaredMethod("c").invoke(null);
    }

    static void c() {
        b();
    }

    static void b() {
        a();
    }
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    static void a() {
        List<StackWalker> stackWalkers =
                List.of(
                    StackWalker.getInstance(),
                    StackWalker.getInstance(SHOW_REFLECT_FRAMES),
                    StackWalker.getInstance(SHOW_HIDDEN_FRAMES)
                );

        stackWalkers.forEach(
            stackWalker -> {
                out.println();
                stackWalker.forEach(out::println);
            }
        );
    }
}
```

执行结果如下：

```
cc.openhome/cc.openhome.StackWalkerDemo3.lambda$a$0(StackWalkerDemo3.java:33)
java.base/java.lang.Iterable.forEach(Iterable.java:75)
cc.openhome/cc.openhome.StackWalkerDemo3.a(StackWalkerDemo3.java:30)
cc.openhome/cc.openhome.StackWalkerDemo3.b(StackWalkerDemo3.java:18)
cc.openhome/cc.openhome.StackWalkerDemo3.c(StackWalkerDemo3.java:14)
cc.openhome/cc.openhome.StackWalkerDemo3.main(StackWalkerDemo3.java:10)

cc.openhome/cc.openhome.StackWalkerDemo3.lambda$a$0(StackWalkerDemo3.java:33)
java.base/java.lang.Iterable.forEach(Iterable.java:75)
cc.openhome/cc.openhome.StackWalkerDemo3.a(StackWalkerDemo3.java:30)
cc.openhome/cc.openhome.StackWalkerDemo3.b(StackWalkerDemo3.java:18)
cc.openhome/cc.openhome.StackWalkerDemo3.c(StackWalkerDemo3.java:14)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccesso
rImpl.java:62)
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMetho
dAccessorImpl.java:43)
java.base/java.lang.reflect.Method.invoke(Method.java:564)
cc.openhome/cc.openhome.StackWalkerDemo3.main(StackWalkerDemo3.java:10)

cc.openhome/cc.openhome.StackWalkerDemo3.lambda$a$0(StackWalkerDemo3.java:33)
cc.openhome/cc.openhome.StackWalkerDemo3$$Lambda$23/25126016.accept(Unknown Source)
java.base/java.lang.Iterable.forEach(Iterable.java:75)
cc.openhome/cc.openhome.StackWalkerDemo3.a(StackWalkerDemo3.java:30)
cc.openhome/cc.openhome.StackWalkerDemo3.b(StackWalkerDemo3.java:18)
cc.openhome/cc.openhome.StackWalkerDemo3.c(StackWalkerDemo3.java:14)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccesso
rImpl.java:62)
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMetho
dAccessorImpl.java:43)
java.base/java.lang.reflect.Method.invoke(Method.java:564)
cc.openhome/cc.openhome.StackWalkerDemo3.main(StackWalkerDemo3.java:10)
```

可以看到默认的堆栈追踪记录最为简洁，而 SHOW_REFLECT_FRAMES 显示了一些反射相关的 Stack Frame，至于 SHOW_HIDDEN_FRAMES 除了显示反射相关的 Stack Frame 之外，还多了个 JVM

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

特定操作的 Stack Frame。

如果需要在 StackWalker.getInstance()时指定多个选项，可以使用 Set 来指定，这时结合 Set.of()会很方便，例如：

```
StackWalker stackWalker = StackWalker.getInstance(
    Set.of(RETAIN_CLASS_REFERENCE, SHOW_REFLECT_FRAMES)
);
```

如果想限制可获取的 Stack Frame 深度，可以使用 StackWalker.getInstance()的另一个版本：

```
StackWalker stackWalker = StackWalker.getInstance(
    Set.of(RETAIN_CLASS_REFERENCE, SHOW_REFLECT_FRAMES),
    10  // 深度最多为 10
);
```

15.6 重点复习

java.util.logging 包提供了日志功能相关类与接口，不必额外配置日志组件，就可在标准 Java 平台使用是其好处。使用日志的起点是 Logger 类，要获取 Logger 实例，必须使用 Logger 的静态方法 getLogger()。

调用 getLogger()时，必须指定 Logger 实例所属名称空间，名称空间以“.”作为层级区分，名称空间层级相同的 Logger，其父 Logger 组态相同。

如果客户端调用了 Logger 实例的 log()方法，首先会以 Level 过滤信息，再看看 Logger 有无设定 Filter 接口的实例。如果有且其 isLoggable()返回 true，才会调用 Handler 实例的 publish()方法，Handler 也可设定自己的 Filter 实例，如果有且其 isLoggable()返回 true，就调用 Formatter 实例的 format()方法格式化信息，最后才调用输出对象将格式化后的信息输出。目前 Logger 的 Handler 处理完，还会传播给父 Logger 的所有 Handler 处理(在通过父 Logger 层级的情况下)。

可以通过 logging.properties 来设定 Logger 组态，启动 JVM 时，指定 java.util.logging.config.file 系统属性为.properties 名称。

应用程序根据不同地区用户，呈现不同语言、日期格式等称为本地化。如果应用程序设计时，可在不修改应用程序情况下，根据不同用户直接采用不同语言、日期格式等，这样的设计考虑称为国际化(Internationalization)，简称 i18n(因为 Internationalization 有 18 个字母)。

国际化的三个重要概念是地区信息、资源包与基础名称(Base Name)。地区信息的对应类是 Locale，ResourceBundle 对象是 JVM 中资源包的代表对象。代表同一组信息但不同地区的各个资源包会共享相同的基础名称，使用 ResourceBundle 的 getBundle()时指定的名称，就是在指定基础名称。

使用 ResourceBundle 时，如何根据基础名称获取对应的信息文档：

(1) 使用指定的 Locale 对象获取信息文档。

(2) 使用 Locale.getDefault()获取的对象获取信息文档。

(3) 使用基础名称获取信息文档。

可以使用 Date 来获取完整日期时间，可单纯使用 toString()获取日期文字描述，或使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

DateFormat 格式化日期。若查看 Date 的 API 文件，会发现许多方法都不再建议使用(Deprecated)，而建议改用 Calendar 的相关方法取代。

规则表示式主要用于字符、字符串格式比较，java.util.regex.Pattern 实例是规则表示式在 JVM 中的代表对象，必须通过 Pattern 的静态方法 compile()来获取，可以使用 matcher()方法指定要比较的字符串，这会返回 java.util.regex.Matcher 实例，表示对指定字符串的比较器。

如果你需要处理大数或者是精度等更复杂的数字问题，就必须对 BigInteger 或 BigDecimal 有更多的认识，若想进行数字的格式化，可以通过 java.text.NumberFormat。

每个线程都会有自己专属的 JVM Stack。JVM Stack 是个先进后出结构，每调用一个方法，JVM 就会建立一个 Stack Frame 来储存局部变量、方法、类等信息并置放至 JVM Stack，方法调用结束后 Stack Frame 就从 JVM Stack 中跳出销毁。

JDK 从 1.4 开始就在 Throwable 与 Thread 类上提供了 getStackTrace()方法，可用来获取当时活动线程的 JVM Stack，并以 StackTraceElement 数组返回 JVM Stack 中全部的 Stack Frame。

为了在堆栈追踪时更方便，JDK9 新增了 Stack-Walking API，具体来说，可以通过 java.lang.StackWalker 的 getInstance()方法，获取 StackWalker 实例。

15.7 课后练习

如果有个 HTML 文档，其中有许多 img 标签，而每个 img 标签都被 a 标签给包裹住。例如：

```
<a href="images/EssentialJavaScript-1-1.png" target="_blank"><img src=
"images/EssentialJavaScript-1-1.png" alt="测试 node 指令" style="max-width:100%;"></a>
```

请撰写程序读取指定的 HTML 文件名，将包含 img 标签的 a 标签去除之后再保存到原文档，也就是执行程序过后，文档中如上的 HTML 要变为：

```
<img src="images/EssentialJavaScript-1-1.png" alt="测试 node 指令" style="max-width:100%;">
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 16 整合数据库

学习目标

- 了解 JDBC 架构
- 使用 JDBC API
- 了解交易与隔离层级
- 认识 RowSet

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

16.1 JDBC 入门

JDBC 是用于执行 SQL 的解决方案，开发人员使用 JDBC 的标准接口，数据库厂商则对接口进行操作，开发人员无须接触底层数据库驱动程序的差异性。在本章中，会介绍一些 JDBC 基本 API 的使用与概念，让读者对 Java 如何存取数据库有所认识。

16.1.1 JDBC 简介

在正式介绍 JDBC 前，先来认识应用程序如何与数据库进行沟通。许多数据库本身是个独立运行的服务器应用程序，用户撰写的应用程序是利用网络通信协议对数据库服务器进行指令交换，以进行数据的增删查找，如图 16.1 所示。

图 16.1 应用程序与数据库利用通信协议沟通

通常你的应用程序会利用一组专门与数据库进行通信协议的链接库，以简化与数据库沟通时的程序撰写，如图 16.2 所示。

图 16.2 应用程序调用链接库以简化程序撰写

问题的重点在于，应用程序如何调用这组链接库？不同的数据库通常会有不同的通信协议，用来联机不同数据库的链接库，在 API 上也会有所不同。如果你的应用程序直接使用这些链接库，例如：

```
XySqlConnection conn = new XySqlConnection("localhost", "root", "1234");
conn.selectDB("gossip");
XySqlQuery query = conn.query("SELECT * FROM USERS");
```

假设这段程序代码中的 API 是某 Xy 数据库厂商链接库所提供，应用程序中要使用到数据库联机时，都会直接调用这些 API，若哪天应用程序打算改用 Ab 厂商数据库及其提供的数据库联机 API，就得修改相关的程序代码。

另外，若 Xy 数据库厂商的链接库底层实际使用了与操作系统相依的功能，若你打算换个操作系统，就还得先考虑一下，是否有提供该平台的数据库程序库。

更换数据库的需求并不是没有，应用程序跨平台也是经常的需求，JDBC 基本上就是用来解决这些问题。JDBC 全名 Java DataBase Connectivity，是 Java 联机数据库的标准规范。具体而言，它定义一组标准类与接口，应用程序需要联机数据库时调用这组标准 API，而标准 API 中的接口会由数据库厂商操作，通常称为 JDBC 驱动程序(Driver)，如图 16.3 所示。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

图 16.3　应用程序调用 JDBC 标准 API

JDBC 标准主要分为两个部分：JDBC 应用程序开发者接口(Application Developer Interface)以及 JDBC 驱动程序开发者接口(Driver Developer Interface)。如果应用程序需要联机数据库，就是调用 JDBC 应用程序开发者接口，相关 API 主要在 **java.sql** 与 **javax.sql** 两个包中，它们在 JDK9 中归于 java.sql 模块，也是本章节的重点；JDBC 驱动程序开发者接口是数据库厂商操作驱动程序时的规范，一般开发者并不用了解，如图 16.4 所示。

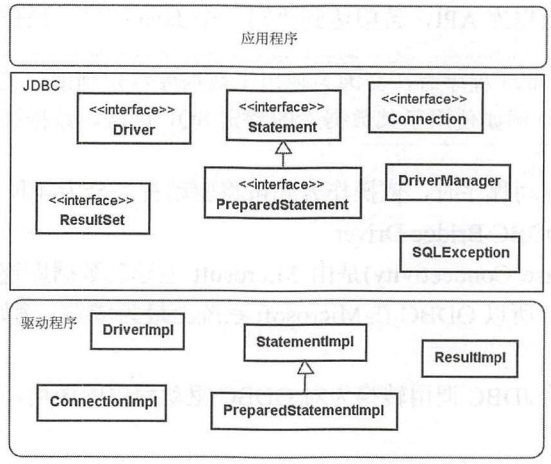


图 16.4　JDBC 应用程序开发者接口

举个例子来说，应用程序会使用 JDBC 联机数据库：

```
Connection conn = DriverManager.getConnection(…);
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM USERS");
```

其中粗体字部分就是标准类(像是 DriverManager)与接口(如 Connection、Statement、ResultSet)等标准 API。假设这段程序代码是联机 MySQL 数据库，则需要在 CLASSPATH 中设定 JDBC 驱动程序，具体来说，就是在 CLASSPATH 中设定一个 JAR 文档，此时应用程序、JDBC 与数据库的关系如图 16.5 所示。

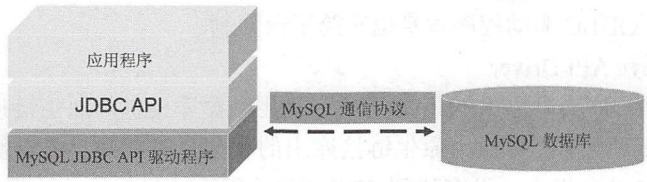


图 16.5　应用程序、JDBC 与数据库的关系

如果将来要换为 Oracle 数据库，只要置换 Oracle 驱动程序。具体来说，就是在 CLASSPATH 改设为 Oracle 驱动程序的 JAR 文档，然而应用程序本身不用修改，如图 16.6 所示。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

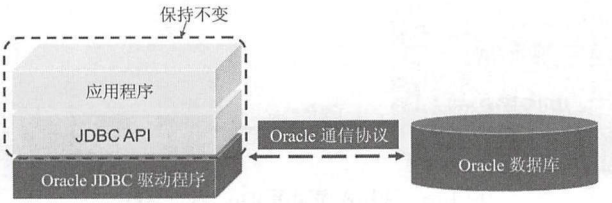


图 16.6　置换驱动程序不用修改应用程序

如果开发应用程序操作数据库时，是通过 JDBC 提供的接口来设计程序，理论上在必须更换数据库时，应用程序无须进行修改，只需要更换数据库驱动程序成果，即可对另一个数据库进行操作。

JDBC 希望达到的目的，是让 Java 程序设计人员在撰写数据库操作程序时，可以有个统一的接口，无须依赖特定数据库 API，希望达到“写一个 Java 程序，操作所有数据库”的目的。

提示 »» 实际上在撰写 Java 程序时，会因为使用了数据库特定功能，而在转移数据库时仍得对程序进行修改，例如使用了某数据库的特定 SQL 语法、数据类型或内部函数调用等。

厂商在操作 JDBC 驱动程序时，依操作方式可将驱动程序分为 4 种类型。

(1) Type 1：JDBC-ODBC Bridge Driver

ODBC(Open DataBase Connectivity)是由 Microsoft 主导的数据库连接标准，基本上 JDBC 是参考 ODBC 制订而来，所以 ODBC 在 Microsoft 系统上最为成熟，例如 Microsoft Access 数据库存取就是使用 ODBC。

Type 1 驱动程序会将 JDBC 调用转换为对 ODBC 驱动程序的调用，由 ODBC 驱动程序操作数据库，如图 16.7 所示。

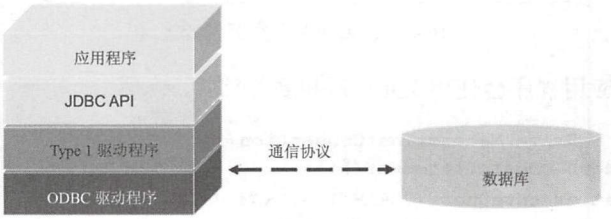


图 16.7　JDBC-ODBC Bridge Driver

由于利用现成的 ODBC 架构，只需要将 JDBC 调用转换为 ODBC 调用，所以要操作这种驱动程序非常简单。不过由于 JDBC 与 ODBC 并非一对一的对应，所以部分调用无法直接转换，因此有些功能受限。而多层调用转换的结果，访问速度也受到限制，ODBC 本身需在平台上先设定好，弹性不足，ODBC 驱动程序本身也有跨平台限制。

(2) Type 2：Native API Driver

这个类型的驱动程序会以原生(Native)方式，调用数据库提供的原生链接库(通常由 C/C++ 操作)，JDBC 的方法调用都会转换为原生链接库中的相关 API 调用。由于使用了原生链接库，所以驱动程序本身与平台相依，没有达到 JDBC 驱动程序的目标之一：跨平台。不过由于直接调用数据库原生 API，因此在速度上，有机会成为 4 种类型中最快的驱动程序，如图 16.8 所示。

非卖品，仅供非商业用途或交流学习使用

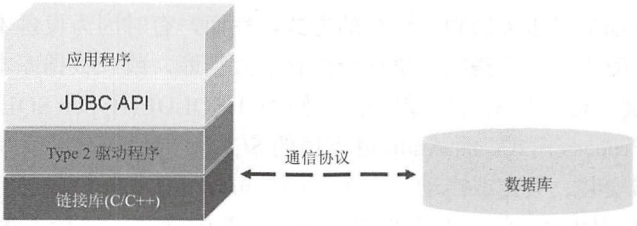


图 16.8 Native API Driver

Type 2 驱动程序有机会成为速度最快的驱动程序，速度的优势是在于获得数据库响应数据后，创建相关 JDBC API 操作对象时。然而驱动程序本身无法跨平台，使用前必须先在各平台进行驱动程序的安装设定(如安装数据库专属的原生链接库)。

(3) Type 3：JDBC-Net Driver

这类型的 JDBC 驱动程序会将 JDBC 方法调用转换为特定的网络协议(Protocol)调用，目的是远程与数据库特定的中介服务器或组件进行协议操作，而中介服务器或组件再真正与数据库进行操作，如图 16.9 所示。

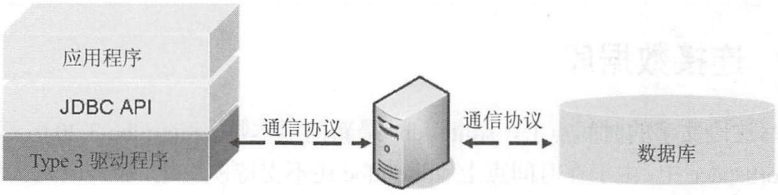


图 16.9 JDBC-Net Driver

由于实际与中介服务器或组件进行沟通时，是利用网络协议的方式，所以客户端这里安装的驱动程序，可以使用纯粹的 Java 技术来实现(基本上就是将 JDBC 调用对应至网络协议而已)，因此这种类型的驱动程序可以跨平台。使用这种类型驱动程序的弹性高，例如可以设计一个中介组件，JDBC 驱动程序与中介组件间的协议是固定的，如果需要更换数据库系统，则只需要更换中介组件，但客户端不受影响，驱动程序也无须更换。但由于通过中介服务器转换，速度较慢，获得架构弹性是使用这种类型驱动程序的目的。

(4) Type 4：Native Protocol Driver

这种类型驱动程序操作通常由数据库厂商直接提供，驱动程序操作会将 JDBC 调用转换为与数据库特定的网络协议，以与数据库进行沟通操作，如图 16.10 所示。

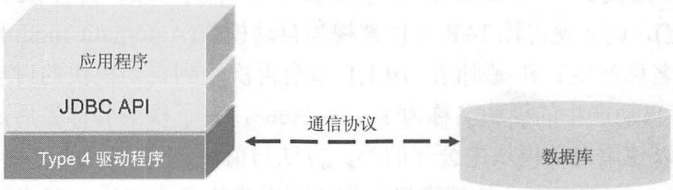


图 16.10 Native Protocol Driver

由于这种类型驱动程序主要的作用，是将 JDBC 调用转换为特定网络协议，所以驱动程序可以使用纯粹 Java 技术实现，因此这种类型驱动程序可以跨平台，在效能上也能有不错的表现。在不需要如 Type 3 获得架构上的弹性时，通常会使用这种类型驱动程序，算是最常见的驱动程序类型。

非卖品，仅供非商业用途或交流学习使用

虽然许多数据库都是采服务器独立运行的方式，然而，有时因为设备本身资源限制，或者是实行测试时方便，应用程序会搭配可执行于内存的数据库，或者数据库本身是个文件，应用程序可直接读/写该文件进行数据的增/删查找。例如 HSQLDB(Hyper SQL Database)就提供了 Memory-Only 与 In-Process 模式，而 Android 支持的 SQLite 则是采直接读/写文件的方式，这类数据库的好处是无须安装、设定或启动，应用程序也可以通过 JDBC 来进行数据库操作。

为了将重点放在 JDBC，免去设定数据库时不必要的麻烦，在接下来的内容中，将使用 SQLite 进行操作，并适当地提示结合 MySQL 时的做法。

sqlite-jdbc 提供了 SQLite 以及它的 JDBC 驱动程序，可以在下面的网址获取最新的 sqlite-jdbc：bitbucket.org/xerial/sqlite-jdbc/downloads/

提示 »» 数据库系统的使用与操作是个很大的主题，本书中并不针对这方面详加探讨，读者请寻找相关的数据库系统相关文件或书籍自行学习，如果对 SQLite 的使用有兴趣，可以参考 SQL As Understood By SQLite，网址如下：
sqlite.org/lang.html
如果对 MySQL 的使用有兴趣，可以参考《MySQL 超新手入门》，网址如下：
www.codedata.com.tw/database/mysql-tutorial-getting-started

16.1.2 连接数据库

在撰写这段文字的时间点上，sqlite-jdbc 最新的版本是 sqlite-jdbc-3.20.0，可将下载的 JAR 存放在 C:\workspace 中，由于在时间点上 sqlite-jdbc 还不支持模块化，那么该怎么使用这个 JAR 呢？

方式之一是采用非模块化的方式，在类路径(使用-cp 指定)包含 JAR 的所在路径。如果程序代码中不会依赖任何 JAR 的类，例如 JDBC 驱动程序类是采用反射(Reflection)来加载，之后通过 JDBC 标准 API 来撰写程序，因此将 sqlite-jdbc 的 JAR 放在类路径上，执行上并没有问题。

然而，本书至今的项目都是采取模块化设计，撰写的类都是在明确定义模块之中，这样的模块称为显式模块(Explicit module)。而第 2 章就介绍过，若是在类路径上包含 JAR，那会使得 JAR 中的类被归类在未命名模块(Unnamed module)，问题就在于，显式模块无法 **requires** 未命名模块，因为未命名模块没有名称，如果程序代码中得使用未命名模块(Unnamed module)的类，编译时期就会找不到类了。

提示 »» 执行时期可以运用反射来存取未命名模块中的类。

若是采用模块化设计，可以将未支持模块化设计的 JAR 文件放在模块路径(使用 --module-path 指定)，这会使得该 JAR 文件被视为自动模块(Automatic module)，自动模块也是命名模块的一种，名称产生有其规则(在 19.1.1 节会再次看到自动模块的讨论)。以 sqlite-jdbc 的 JAR 文件为例，自动产生的模块名称为 sqlite.jdbc，有了模块名称之后，就可以 requires 自动模块，也就可以使用自动模块中公开的类、方法与值域。

如果无法从 JAR 文件名中，依规则自动生成正确模块名称的话，被放到模块路径的 JAR 文件会导致执行时产生错误信息。

提示 »» 可以使用 jar 的--describe-module 来查看自动模块名称。

基于兼容性，自动模块有隐含的模块定义，可以读取其他模块，其他模块也可以存取(与深层反射)自动模块。应用程序在迁移至模块化设计的过程中，自动模块会是未具名模块至显式模

非卖品，仅供非商业用途或交流学习使用

块之间的桥梁。

如果使用 IDE，程序项目会有管理类路径与模块路径的方式，通常是“新增 JAR”之类的指令。例如 NetBeans 是模块化项目的话，可以新增链接库：

(1) 在项目上的“库”节点上右击，从弹出的快捷菜单中选择“添加 JAR/文件夹”命令。

(2) 在出现的“添加 JAR/文件夹”对话框中，选择驱动程序 JAR 文件后单击“打开”按钮。

(3) 确认项目的“库”节点上出现 JAR 文件，NetBeans IDE 会将未支持模块化的 JAR 视为自动模块，也就是会放在模块路径之中，并自动在 module-info.java 中 `requires` 自动模块名称。本章接下来的范例项目，都是采取此方式，将 sqlite-jdbc 的 JAR 文件放在模块路径上。

提示 »» IDE 也可以管理常用的 JAR 文档(有的 JAR 文件会内置)，例如在 NetBeans 的“库”节点上右击，在弹出的快捷菜单中选择“添加库”命令，打开“添加库”对话框，此时会出现 NetBeans 已管理(或内建)的常用 JAR，其中就内置了 MySQL 驱动程序，如图 16.11 所示。

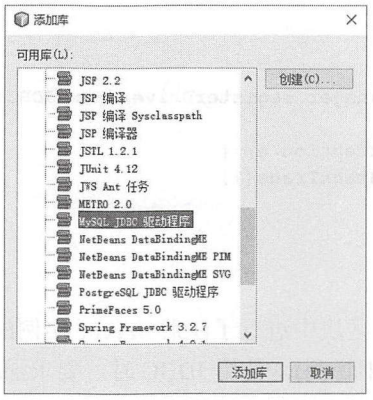


图 16.11　管理常用的 JAR

基本数据库操作相关的 JDBC 接口或类是位于 **`java.sql`** 包中。而这个套件在 JDK9 中是位于 **`java.sql`** 模块之中，如果采用模块化设计要在模块的模块描述文件中加入 **`requires java.sql`**。

在编写程序时，要取得数据库联机，必须有几个动作：

- 注册 `Driver` 操作对象。
- 取得 `Connection` 操作对象。
- 关闭 `Connection` 操作对象。

1. 注册 `Driver` 操作对象

操作 `Driver` 接口的对象是 JDBC 进行数据库存取的起点，以 SQLite 操作的驱动程序为例，`org.sqlite.JDBC` 类操作了 **`java.sql.Driver`** 接口，管理 `Driver` 操作对象的类是 **`java.sql.DriverManager`**。基本上，必须调用其静态方法 **`registerDriver()`** 进行注册：

```
DriverManager.registerDriver(new org.sqlite.JDBC());
```

不过实际上很少自行撰写程序代码进行这个动作，只要想办法加载 **`Driver`** 接口的操作类.class 文档，就会完成注册。例如，可以通过 **`java.lang.Class`** 类的 **`forName()`**(第 17 章会详细说明这个方法)，动态加载驱动程序类：

非卖品，仅供非商业用途或交流学习使用

```
try {
    Class.forName("org.sqlite.JDBC");
}
catch(ClassNotFoundException e) {
    throw new RuntimeException("找不到指定的类");
}
```

如果查看 SQLite 的 org.sqlite.JDBC 类操作原始码：

```
package org.sqlite;

import java.sql.Driver;
import java.sql.DriverManager;
...

public class JDBC implements Driver
{
    public static final String PREFIX = "jdbc:sqlite:";

    static {
        try {
            DriverManager.registerDriver(new JDBC());
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
    ...
}
```

可以发现，在 static 区块中进行了注册 Driver 实例的动作，而 static 区块会在载入.class 文档时执行(第 17 章会详细说明)。使用 JDBC 时，要求加载.class 文档的方式有 4 种：

(1) 使用 Class.forName()。

(2) 自行建立 Driver 接口操作类的实例。

(3) 启动 JVM 时指定 jdbc.drivers 属性。

(4) 设定 JAR 中/META-INF/services/java.sql.Driver 文档。

第一种方式刚才已经说明。第二种方式就是直接撰写程序代码：

```
java.sql.Driver driver = new org.sqlite.JDBC();
```

由于要建立对象，基本上就要加载.class 文档，自然也就会执行类的静态区块完成驱动程序注册。第三种方式就是执行 java 指令时如下：

```
> java -Djdbc.drivers=org.sqlite.JDBC;ooo.XXXDriver 其他选项… YourProgram
```

应用程序可能同时联机多个厂商的数据库，所以 DriverManager 也可以注册多个驱动程序实例，以上方式如果需要指定多个驱动程序类，就用分号分隔。第四种方式则是 JDK6 之后 JDBC 4.0 新特性，只要在驱动程序操作的 JAR 文档/META-INF/services 文件夹中，放置一个 java.sql.Driver 文档，当中撰写 Driver 接口的操作类名称全名，DriverManager 会自动读取这个文档并找到指定类进行注册。

2. 取得 Connection 操作对象

Connection 接口的操作对象是数据库联机代表对象，要取得 Connection 操作对象，可以通过 DriverManager 的 **getConnection()**：

非卖品，仅供非商业用途或交流学习使用

```
Connection conn = DriverManager.getConnection(jdbcUrl);
```

在调用 getConnection()时，必须提供 JDBC URL，其定义了连接数据库时的协议、子协议、数据源识别：

```
协议:子协议:数据源识别
```

除了“协议”在 JDBC URL 中总是 jdbc 开始之外，JDBC URL 格式各个数据库都不相同，必须查询数据库产品使用手册。以下以 SQLite 为例，“子协议”是桥接的驱动程序、数据库产品名称或联机机制，例如使用 SQLite 的话，子协议名称是 sqlite。“数据源识别”可能是数据库的地址、端口号、名称、用户、密码等信息。举个例子来说，SQLite 的 JDBC URL 撰写方式如下：

```
jdbc:sqlite:demo.sqlite
```

这表示数据库名称会是 demo.sqlite，由于 SQLite 直接使用文件作为数据库储存之用，因此 demo.sqlite 也就表示储存用的文件名。

提示 »» 在 MySQL 中，你可以在 Command Line Client 中使用以下指令来建立数据库：

CREATE schema demo;

MySQL 的 JDBC URL 撰写方式如下：

Jdbc:mysql://主机名:端口/数据库名称?参数=值&参数=值

主机名可以是本机(localhost)或其他联机主机名、地址，MySQL 端口默认为 3306。例如要连接 demo 数据库，并指明用户名称与密码，可以这样指定：

```
jdbc:mysql://localhost:3306/demo?user=root&password=openhome
```

如果要使用中文存取的话，还必须给定参数 useUnicode 及 characterEncoding，表明是否使用 Unicode，并指定字符编码方式。例如(假设数据库表格编码使用 UTF8)：

```
jdbc:mysql://localhost:3306/demo?user=root&password =123&useUnicode=
true&characterEncoding=UTF8
```

有的时候会将 JDBC URL 撰写在 XML 配置文件中，此时不能直接在 XML 中写&符号，而必须改写为&替代字符。例如：

```
jdbc:mysql://localhost:3306/demo?user=root&password=123&useUnicode
=true&amp;characterEncoding=UTF8
```

如果要直接通过 DriverManager 的 getConnection()连接数据库，完整的代码段如下：

```
String jdbcUrl =  "jdbc:sqlite:demo.sqlite";
Connection conn = null;
SQLException ex = null;
try {
    conn = DriverManager.getConnection(jdbcUrl);
    ...
}
catch(SQLException e) {
    ex = e;
}
finally {
    if(conn != null) {
        try {
            conn.close();
        }
        catch(SQLException e) {
            if(ex == null) {
                ex = e;
            }
```


非卖品，仅供非商业用途或交流学习使用

```
                else {
                    ex.addSuppressed(e);
                }
            }
        }
        if(ex != null) {
            throw new RuntimeException(ex);
        }
    }
```

SQLException 是在处理 JDBC 时常遇到的异常对象，为数据库操作过程发生错误时的代表对象。SQLException 是受检异常(Checked Exception)，必须使用 try...catch...finally 明确处理，在异常发生时尝试关闭相关资源。

提示>>> SQLException 有个子类 **SQLWarning**，如果数据库执行过程中发生了一些警示信息，会建立 SQLWarning 但不会抛出(throw)，而是以链接方式收集起来。可以使用 **Connection**、**Statement**、**ResultSet** 的 **getWarnings()** 来取得第一个 SQLWarning，使用这个对象的 **getNextWarning()** 可以取得下一个 SQLWarning，由于它是 SQLException 的子类，所以必要时也可当作异常抛出。

3. 关闭 Connection 操作对象

取得 Connection 对象之后，可以使用 **isClosed()** 方法测试与数据库的连接是否关闭。在操作完数据库之后，若确定不再需要连接，则必须使用 **close()** 来关闭与数据库的连接，以释放连接时相关的必要资源，如联机相关对象、授权资源等。

除了像前一个范例代码段，自行撰写 try...catch...finally 尝试关闭 Connection 之外，从 JDK7 之后，JDBC 的 Connection、Statement、ResultSet 等接口都是 java.lang.AutoCloseable 子接口，因此可以使用尝试自动关闭资源语法来简化程序撰写。例如前一个程序片段，可以进行以下简化：

```
String jdbcUrl = "jdbc:sqlite:demo.sqlite";
try(Connection conn = DriverManager.getConnection(jdbcUrl)) {
    ...
}
catch(SQLException e) {
    throw new RuntimeException(e);
}
```

以上是撰写程序上的一些简介，然而在底层，DriverManager 如何进行联机呢？DriverManager 会在循环中逐一取出注册的每个 Driver 实例，使用指定的 JDBC URL 来调用 Driver 的 connect() 方法，尝试取得 Connection 实例。以下是 DriverManager 中相关原始码的重点节录：

```
SQLException reason = null;
for (int i = 0; i < drivers.size(); i++) {                    // 逐一取得 Driver 实例
    ...
    DriverInfo di = (DriverInfo)drivers.elementAt(i);
    ...
    try {
        Connection result = di.driver.connect(url, info);// 尝试联机
        if (result != null) {
            return (result);                                  // 取得 Connection 就返回
        }
    } catch (SQLException ex) {
        if (reason == null) {                                 // 记录第一个发生的异常
```


非卖品，仅供非商业用途或交流学习使用

```
                    reason = ex;
                }
            }
        }
        if (reason != null)     {
            println("getConnection failed: " + reason);
            throw reason;                                    // 如果有异常对象就抛出
        }
        throw new SQLException(    // 没有适用的 Driver 实例，抛出异常
                "No suitable driver found for "+ url, "08001");
```

Driver 的 connect() 方法在无法取得 Connection 时会返回 null，所以简单来说，DriverManager 就是逐一使用 Driver 实例尝试联机。如果联机成功就返回 Connection 对象，如果当中有异常发生，DriverManager 会记录第一个异常，并继续尝试其他的 Driver，在所有 Driver 都试过了也无法取得联机，若原先尝试过程中有记录异常就抛出，没有的话，也是抛出异常告知没有适合的驱动程序。

以下是一个简单范例，测试一下可否联机数据库并取得 Connection 实例。

JDBCDemo ConnectionDemo.java

```
package cc.openhome;

import java.sql.*;
import static java.lang.System.out;

public class ConnectionDemo {
    public static void main(String[] args)
                            throws ClassNotFoundException, SQLException {
        String jdbcUrl = "jdbc:sqlite:demo.sqlite ";
        try(Connection conn =                                    取得 Connection 对象
                DriverManager.getConnection(jdbcUrl, user, passwd)) {
            out.printf("已%s 数据库联机%n",conn.isClosed() ? "关闭" : "开启");
        }

    }
}
```

这个范例对 Connection 使用尝试自动关闭资源语法，所以执行完 try 区块后，Connection 的 close() 就会被调用。如果顺利取得联机，程序执行结果如下：

```
已开启数据库联机
```

由于 SQLite 使用文件来作为数据库储存之用，执行此范例时若 demo.sqlite 文件不存在，会在项目文件夹中自动建立文件，若想删除数据库，只要将该文件删除就可以了。

提示 »» 实际上很少直接从 DriverManager 中取得 Connection。想想看，如果你在设计 API，用户无法提供 JDBC URL(商业数据库还需名称、密码等敏感信息)，你要怎么取得 Connection？答案是通过稍后要介绍的 javax.sql.DataSource。

16.1.3 使用 Statement、ResultSet

Connection 是数据库连接的代表对象，接下来要执行 SQL 的话，必须取得 **java.sql.Statement** 操作对象，它是 SQL 描述的代表对象。可以使用 Connection 的 **createStatement()** 建

非卖品，仅供非商业用途或交流学习使用

立 Statement 对象：

```
Statement stmt = conn.createStatement();
```

取得 Statement 对象之后，可以使用 **executeUpdate()**、**executeQuery()** 等方法来执行 SQL。executeUpdate() 主要用来执行 CREATE TABLE、INSERT、DROP TABLE、ALTER TABLE 等会改变数据库内容的 SQL。

例如，若想在 demo.sqlite 数据库中建立一个 messages 表格，可以如下使用 Statement 的 executeUpdate() 方法。

JDBCDemo StatementDemo.java

```
package cc.openhome;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class StatementDemo {
    public static void main(String[] args) {
        String jdbcUrl = "jdbc:sqlite:demo.sqlite";
        try(Connection conn = DriverManager.getConnection(jdbcUrl);
            Statement statement = conn.createStatement()) {
            statement.executeUpdate(
                "CREATE TABLE messages (" +
                    "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL," +
                    "name TEXT NOT NULL," +
                    "email TEXT NOT NULL," +
                    "msg TEXT NOT NULL)"
            );
            System.out.println("建立 messages 表格了");
        } catch(SQLException ex) {
            throw new RuntimeException(ex);
        }
    }
}
```

提示 »» 在 MySQL 中，可以直接在 Command Line Client 中使用以下指令，在 demo 数据库中建立表格：

```
USE demo;
CREATE TABLE messages (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(20) NOT NULL,
    email CHAR(40),
    msg TEXT NOT NULL
) CHARSET=UTF8;
```

在建立表格之后，如果要在这个表格中插入一笔数据，可以这样使用 Statement 的 executeUpdate() 方法：

```
stmt.executeUpdate("INSERT INTO t_messages VALUES(1, 'justin', " +
        "'justin@mail.com', 'mesage...')");
```

Statement 的 executeQuery() 方法用于 SELECT 等查询数据库的 SQL，executeUpdate() 会返回 **int** 结果，表示数据变动的笔数，executeQuery() 会返回 **java.sql.ResultSet** 对象，代表查询

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

结果，查询结果会是一笔一笔的数据。可以使用 ResultSet 的 **next()** 移动至下一笔数据，它会返回 true 或 false 表示是否有下一笔数据。接着可以使用 getXXX() 取得数据，如 **getString()**、**getInt()**、**getFloat()**、**getDouble()** 等方法，分别取得相对应的字段类型数据。**getXXX()** 方法都提供按域名或是字段顺序取得数据的方法。下面的例子，指定域名来取得数据：

```
ResultSet result = stmt.executeQuery("SELECT * FROM t_messages");
while(result.next()) {
    int id = result.getInt("id");
    String name = result.getString("name");
    String email = result.getString("email");
    String msg = result.getString("msg");
    // ...
}
```

使用查询结果字段顺序来显示结果的方式如下(注意索引是从 1 开始)：

```
ResultSet result = stmt.executeQuery("SELECT * FROM t_messages");
while(result.next()) {
    int id = result.getInt(1);
    String name = result.getString(2);
    String email = result.getString(3);
    String msg = result.getString(4);
    // ...
}
```

Statement 的 execute() 可以用来执行 SQL，并可以测试 SQL 是执行查询或更新，返回 true 表示 SQL 执行将返回 ResultSet 作为查询结果，此时可以使用 getResultSet() 取得 ResultSet 对象。如果 execute() 返回 false，表示 SQL 执行会返回更新笔数或没有结果，此时可以使用 getUpdateCount() 取得更新笔数。如果事先无法得知 SQL 是进行查询或更新，就可以使用 execute()。例如：

```
if(stmt.execute(sql)) {
    ResultSet rs = stmt.getResultSet();   // 取得查询结果 ResultSet
    ...
}
else {                                    // 这是个更新操作
    int updated = stmt.getUpdateCount();  // 取得更新笔数
    ...
}
```

视需求而定，Statement 或 ResultSet 在不使用时，可以使用 close() 将之关闭，以释放相关资源。Statement 关闭时，所关联的 ResultSet 也会自动关闭。

接下来以一个简单的留言板作为示范，制作一个 MessageDAO 来存取数据库。

JDBCDemo MessageDAO.java

```
package cc.openhome;

import java.sql.*;
import java.util.*;

public class MessageDAO {
    private String url;
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
    private String user;
    private String passwd;

    public MessageDAO(String url, String user, String passwd) {
        this.url = url;
        this.user = user;
        this.passwd = passwd;
    }
                                ❶这个方法会在数据库中新增留言
                                ↓
    public void add(Message message) {
                                                ❷取得 Connection 对象
                                                ↓
        try(Connection conn = DriverManager.getConnection(url, user, passwd);
            Statement statement = conn.createStatement()) {  ←❸建立 Statement 对象
            String sql = String.format(
                  "INSERT INTO messages(name, email, msg) VALUES ('%s', '%s', '%s')",
                  message.getName(), message.getEmail(), message.getMsg());
            statement.executeUpdate(sql);  ←❹执行 SQL 描述句
        } catch(SQLException ex) {
            throw new RuntimeException(ex);
        }
    }

    public List<Message> get() {  ←❺这个方法会从数据库中查询所有留言
        List<Message> messages = new ArrayList<>();
        try(Connection conn = DriverManager.getConnection(url, user, passwd);
            Statement statement = conn.createStatement()) {
            ResultSet result =
                    statement.executeQuery("SELECT * FROM messages");
            while (result.next()) {
                Message message = toMessage(result);
                messages.add(message);
            }
        } catch(SQLException ex) {
            throw new RuntimeException(ex);
        }
        return messages;
    }

    private Message toMessage(ResultSet result) throws SQLException {
        Message message = new Message();
        message.setId(result.getLong(1));
        message.setName(result.getString(2));
        message.setEmail(result.getString(3));
        message.setMsg(result.getString(4));
        return message;
    }
}
```


严禁网络传播本PDF，违者责任自负！

这个对象会从 DriverManager 取得 Connection 对象❷。add()接受一个 Message 对象❶，操作中在数据库中利用 Statement 对象❸，执行 SQL 描述新增一笔留言❹。get()会从数据库中取回所有留言，并放在一个 List<Message>对象中返回❺。

提示 »» JDBC 规范提到关闭 Connection 时，会关闭相关资源，但没有明确说明是哪些相关资源。通常驱动程序操作时，会在关闭 Connection 时，一并关闭关联的 Statement，但最好留意是否真的关闭了资源，自行关闭 Statement 是比较保险的做法。以上范例对 Connection 与 Statement 使用了尝试自动关闭资源语法。

范例中的 Message 只是用来封装留言信息的简单类：

JDBCDemo Message.java

```
package cc.openhome;

import java.io.Serializable;

public class Message implements Serializable {
    private Long id;
    private String name;
    private String email;
    private String msg;

    public Message() {}
    public Message(String name, String email, String msg) {
        this.name = name;
        this.email = email;
        this.msg = msg;
    }

    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }

    public String getMsg() {
        return msg;
    }
    public void setMsg(String msg) {
        this.msg = msg;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```


严禁网络传播本PDF，违者责任自负！

下例撰写一个简单的 MessageDAODemo 类来使用 MessageDAO。

JDBCDemo MessageDAODemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.Scanner;

public class MessageDAODemo {
    public static void main(String[] args) throws Exception {
        MessageDAO dao = new MessageDAO("jdbc:sqlite:demo.sqlite");
        // 在 Windows 的 NetBeans 中，其 Output 窗口的标准输入编码是 MS950
        Scanner console = new Scanner(System.in, " MS950");
        while(true) {
            out.print("(1) 显示留言 (2) 新增留言：");
            switch(Integer.parseInt(console.nextLine())) {
                case 1:
                    dao.get().forEach(message -> {
                        out.printf("%d\t%s\t%s\t%s%n",
                                message.getId(),
                                message.getName(),
                                message.getEmail(),
                                message.getMsg());
                    });
                    break;
                case 2:
                    out.print("姓名：");
                    String name = console.nextLine();
                    out.print("邮件：");
                    String email = console.nextLine();
                    out.print("留言：");
                    String msg = console.nextLine();
                    dao.add(new Message(name, email, msg));
            }
        }
    }
}
```

以下是个执行的范例：

```
(1) 显示留言 (2) 新增留言：2
姓名：良葛格
邮件：caterpillar@openhome.cc
留言：这是一篇测试留言！
(1) 显示留言 (2) 新增留言：1
1       良葛格      caterpillar@openhome.cc这是一篇测试留言！
```

注意 »» 范例中怎么没有用 Class.forName() 载入 Driver 操作类？别忘了，JDK6 之后支持 JDBC 4.0，只要驱动程序 JAR 中有/META-INF/services/java.sql.Driver 文档，就会自动读取当中设定的 Driver 操作类。

16.1.4 使用 PreparedStatement、CallableStatement

Statement 在执行 executeQuery()、executeUpdate() 等方法时，如果有些部分是动态的数据，

严禁网络传播本PDF，违者责任自负！

必须使用+运算符串接字符串以组成完整的 SQL 语句，十分不方便。例如前面范例中在新增留言时，必须这样串接 SQL 语句：

```
Statement statement = conn.createStatement();
String sql = String.format(
        "INSERT INTO messages(name, email, msg) VALUES ('%s', '%s', '%s')",
        message.getName(), message.getEmail(), message.getMsg());
statement.executeUpdate(sql);
```

如果有些操作只是 SQL 语句当中某些参数有所不同，其余的 SQL 子句皆相同，则可以使用 **java.sql.PreparedStatement**。使用 Connection 的 **prepareStatement()**方法建立好预先编译(Precompile)的 SQL 语句，当中参数会变动的部分，先指定“?”这个占位字符。例如：

```
PreparedStatement stmt = conn.prepareStatement(
                    "INSERT INTO messages VALUES(?, ?, ?, ?)");
```

等到需要真正指定参数执行时，再使用相对应的 setInt()、setString()等方法，指定“?”处真正应该有的参数。例如：

```
stmt.setInt(1, 2);
stmt.setString(2, "momor");
stmt.setString(3, "momor@mail.com");
stmt.setString(4, "message2...");
stmt.executeUpdate();
stmt.clearParameters();
```

要让 SQL 执行生效，执行 **executeUpdate()**或 **executeQuery()**方法(如果是查询的话)。在这次的 SQL 执行完毕后，可以调用 **clearParameters()**清除设置的参数，之后就能再次使用这个 PreparedStatement 实例，所以使用 PreparedStatement，可以先准备好一段 SQL，并重复使用这段 SQL 语句。

可以使用 PreparedStatement 改写前面 MessageDAO 中 add()执行 SQL 语句的部分。例如：

JDBCDemo MessageDAO2.java

```
package cc.openhome;

import java.sql.*;
import java.util.*;

public class MessageDAO2 {
    ...
    public void add(Message message) {
        try(Connection conn = DriverManager.getConnection(url, user, passwd);
            PreparedStatement statement = conn.prepareStatement(
                    "INSERT INTO messages(name, email, msg) VALUES (?,?,?)")) {
            statement.setString(1, message.getName());
            statement.setString(2, message.getEmail());
            statement.setString(3, message.getMsg());
            statement.executeUpdate();
        } catch(SQLException ex) {
            throw new RuntimeException(ex);
        }
    }
    ...
}
```


严禁网络传播本PDF，违者责任自负！

这样的写法显然比串接 SQL 的方式好得多。不过，使用 PreparedStatement 的好处不仅如此，之前提过，在这次的 SQL 执行完毕后，可以调用 clearParameters()清除设置的参数，之后就可以再次使用这个 PreparedStatement 实例。也就是说必要的话，可以考虑制作描述句池(Statement Pool)将一些频繁使用的 PreparedStatement 重复使用，减少生成对象的负担。

在驱动程序支持的情况下，使用 PreparedStatement，可以将 SQL 描述预编译为数据库的执行指令。由于已经是数据库的可执行指令，执行速度可以快许多。例如有些纯 Java 数据库，其驱动程序可以将 SQL 预编译为位码(byte code)格式，在 JVM 中执行就快许多了，而不像 Statement 对象，是在执行时将 SQL 直接送到数据库，由数据库做剖析、直译再执行。

使用 PreparedStatement 在安全上也可以有点贡献。举个例子来说，如果原先使用串接字符串的方式来执行 SQL：

```
Statement statement = connection.createStatement();
String queryString = "SELECT * FROM user_table WHERE username='" +
        username + "' AND password='" + password + "'";
ResultSet resultSet = statement.executeQuery(queryString);
```

其中 username 与 password 若是来自用户的输入字符串，原本是希望用户输入名称密码，组合之后的 SQL 应该像是这样：

```
SELECT * FROM user_table
    WHERE username='caterpillar' AND password='openhome'
```

但如果用户在密码的部分，输入了“' OR '1'='1”这样的字符串，而又没有针对用户的输入进行字符检查过滤动作，这个奇怪的字符串最后组合出来的 SQL 会如下：

```
SELECT * FROM user_table
    WHERE username='caterpillar' AND password='' OR '1'='1'
```

方框是密码请求参数的部分，将方框拿掉会更清楚地看出这个 SQL 有什么问题！

```
SELECT * FROM user_table
    WHERE username='caterpillar' AND password='' OR '1'='1'
```

AND 子句之后的判断式永远成立，也就是说，用户不用输入正确的密码，也可以查询出所有的数据，这就是 SQL Injection 的简单例子。

以串接或 String.format()的方式组合 SQL 描述，基本上就会有 SQL Injection 的隐患，如果这样改用 PreparedStatement 的话：

```
PreparedStatement stmt = conn.prepareStatement(
    "SELECT * FROM user_table WHERE username=? AND password=?");
stmt.setString(1, username);
stmt.setString(2, password);
```

在这里 username 与 password 将被视作是 SQL 中纯粹的字符串，而不会被当作 SQL 语法来解释，所以就可避免这个例子的 SQL Injection 问题。

其实问题不仅是在串接或格式化字符串时比较麻烦，以及 SQL Injection 发生的可能性较大。由于串接或格式化字符串会产生新的 String 对象，如果串接字符串动作经常进行(例如在循环中进行 SQL 串接的动作)，那会是效能负担上的隐患。

如果撰写数据库的预存程序(Stored Procedure)，并想使用 JDBC 来调用，则可使用 **java.sql.CallableStatement**。调用的基本语法如下：

严禁网络传播本PDF，违者责任自负！

```
{?= call <程序名称>[<自变量 1>,<自变量 2>, ...]}
{call <程序名称>[<自变量 1>,<自变量 2>, ...]}
```

CallableStatement 的 API 使用，基本上与 PreparedStatement 差别不大，除了必须调用 **prepareCall()** 建立 CallableStatement 异常外，一样是使用 setXXX() 设定参数，如果是查询操作，使用 executeQuery()，如果是更新操作，使用 executeUpdate()。另外，可以使用 **registerOutParameter()** 注册输出参数等。

提示 »» 使用 JDBC 的 CallableStatement 调用预存程序，重点在于了解各个数据库的预存程序如何撰写及相关事宜，用 JDBC 调用预存程序，也表示应用程序将与数据库产生直接的相依性。

在使用 PreparedStatement 或 CallableStatement 时，必须注意 SQL 类型与 Java 数据类型的对应，因为两者本身并不是一对一对应，java.sql.Types 定义了一些常数代表 SQL 类型。表 16.1 所示为 JDBC 规范建议的 SQL 类型与 Java 类型的对应。

表 16.1 Java 类型与 SQL 类型的对应

Java 类型	SQL 类型
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	FLOAT
double	DOUBLE
byte[]	BINARY、VARBINARY、LONGBINARY
java.lang.String	CHAR、VARCHAR、LONGVARCHAR
java.math.BigDecimal	NUMERIC、DECIMAL
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

要注意的是，日期时间在 JDBC 中并不是使用 java.util.Date，这个对象可代表的日期时间格式是“年、月、日、时、分、秒、毫秒”。在 JDBC 中要表示日期，是使用 **java.sql.Date**，其日期格式是“年、月、日”，要表示时间的话则是使用 **java.sql.Time**，其时间格式为“时、分、秒”，如果要表示“时、分、秒、微秒”的格式，则使用 **java.sql.Timestamp**。

在 13.2 节介绍过 JDK8 新时间日期 API，对于 TimeStamp 实例，可以使用 **toInstant()** 方法将之转为 Instant 实例，如果有个 Instant 实例，可以通过 TimeStamp 的 **from()** 静态方法，将之转为 TimeStamp 实例。例如：

```
Instant instant = timeStamp.toInstant();
Timestamp timestamp2 = Timestamp.from(instant);
```


严禁网络传播本PDF，违者责任自负！

16.2 JDBC 进阶

上一节介绍了 JDBC 入门概念与相关 API，在这一节，将说明更多进阶 API 的使用，如使用 DataSource 取得 Connection，使用 PreparedStatement、ResultSet 进行更新操作等。

16.2.1 使用 DataSource 取得联机

商业数据库在联机时，基本上必须提供 JDBC URL、用户名称、密码等信息(DriverManager 上有个 getConnection()方法可接受 JDBC URL、用户名称、密码)。然而在实际应用程序开发时，JDBC URL、使用者名称、密码等是很敏感的信息，如果 MessageDAO 的开发者无法得知这些信息，该如何改写 MessageDAO?

答案是可以让 MessageDAO 依赖于 **javax.sql.DataSource** 接口，通过其定义的 **getConnection()** 方法取得 Connection。例如：

JDBCDemo MessageDAO3.java

```
package cc.openhome;

import java.sql.*;
import java.util.*;
import javax.sql.DataSource;

public class MessageDAO3 {
    private DataSource dataSource;
    public MessageDAO3(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void add(Message message) {
        try(Connection conn = dataSource.getConnection();
            PreparedStatement statement = conn.prepareStatement(
                  "INSERT INTO messages(name, email, msg) VALUES (?,?,?)")) {
            ...
        } catch(SQLException ex) {
            throw new RuntimeException(ex);
        }
    }

    public List<Message> get() {
        List<Message> messages = null;
        try(Connection conn =  dataSource.getConnection();
            Statement statement = conn.createStatement()) {
            ...
        } catch(SQLException ex) {
            throw new RuntimeException(ex);
        }
        return messages;
    }
}
```

单看这个 MessageDAO3，并不会知道 DataSource 操作对象是从哪个 URL、使用哪个名称、密码、内部如何建立 Connection，日后要修改数据库服务器主机位置，或者是为了打算重复利

用 Connection 对象而想要加入联机池(Connection Pool)机制等情况，这个 MessageDAO3 都不用修改。

提示 »» 对于服务器形式的数据库，要取得数据库联机，必须打开网络联机(中间经过实体网络)，连接至数据库服务器后，进行协议交换(也就是数次的网络数据往来)以进行验证名称、密码等确认动作。也就是取得数据库联机事件耗时间及资源的动作。尽量利用已打开的联机，也就是重复利用取得的 Connection 实例，是改善数据库联机效能的一个方式。采用联机池是基本做法。

以下范例操作具有简单连接池的 DataSource，示范如何重复使用已取得的 Connection：

JDBCDemo SimpleConnectionPoolDataSource.java

```
package cc.openhome;

import java.util.*;
import java.io.*;
import java.sql.*;
import java.util.concurrent.Executor;
import java.util.logging.Logger;
import javax.sql.DataSource;                                    ❶操作 DataSource

public class SimpleConnectionPoolDataSource implements DataSource {
    private Properties props;
    private String url;
    private int max; // 连接池中最大 Connection 数目
    private List<Connection> conns; ←— ❷维护可重用的 Connection 对象

    public SimpleConnectionPoolDataSource()
            throws IOException, ClassNotFoundException {
        this("jdbc.properties");
    }                                          ❸可指定.properties 文档

    public SimpleConnectionPoolDataSource(String configFile)
                        throws IOException, ClassNotFoundException {
        props = new Properties();
        props.load(new FileInputStream(configFile));

        url = props.getProperty("cc.openhome.url");
        max = Integer.parseInt(props.getProperty("cc.openhome.poolmax"));

        conns = Collections.synchronizedList(new ArrayList<>());
    }

    public synchronized Connection getConnection() throws SQLException {
        if(conns.isEmpty()) { ←— ❹如果 List 为空就建立新的 ConnectionWrapper
            return new ConnectionWrapper(
                    DriverManager.getConnection(url),
                    conns,
                    max
            );
        }
        else { ←— ❺否则返回 List 中一个 Connection
            return conns.remove(conns.size() - 1);
        }
    }                                          ❻ConnectionWrapper 操作 Connection 界

    private class ConnectionWrapper implements Connection {
        private Connection conn;
        private List<Connection> conns;
```

```
        private int max;

        public ConnectionWrapper(Connection conn,List<Connection> conns, int max) {

            this.conn = conn;
            this.conns = conns;
            this.max = max;
        }

        @Override
        public void close() throws SQLException {
            if(conns.size() == max) { ←— ❼ 如果超出最大可维护 Connection 数
                conn.close();                量就关闭 Connection
            }
            else {
                conns.add(this); ←— ❽ 否则放入 List 中以备重用
            }
        }

        @Override
        public Statement createStatement() throws SQLException {
            return conn.createStatement();
        }
        ...
    }
    ...
}
```

SimpleConnectionPoolDataSource 操作了 DataSource 接口❶，其中使用 List<Connection>实例维护可重用的 Connection❷，联机相关信息可以使用.properties 设定❸。如果客户端调用 getConnection()方法尝试取得联机，若 List<Connection>为空，则建立新的 Connection 并打包在 ConnectionWrapper 后返回❹，如果不为空就直接从 List<Connection>移出返回❺。

ConnectionWrapper 操作了 Connection 接口❻，大部分方法操作时都是直接委托给被打包的 Connection 实例。ConnectionWrapper 操作 close()方法时，会看看维护 Connection 的 List<Connection>容量是否到了最大值，如果是就直接关闭被打包的 Connection❼，否则就将自己置入 List<Connection>以备重用❽。

如果准备一个 jdbc.properties，代码如下：

JDBCDemo jdbc.properties

```
cc.openhome.url=jdbc:sqlite:demo.sqlite
cc.openhome.poolmax=10
```

那么就可以这样使用 SimpleConnectionPoolDataSource 与 MessageDAO3：

JDBCDemo MessageDAODemo3.java

```
package cc.openhome;
import java.util.Scanner;
public class MessageDAODemo3 {
    public static void main(String[] args) throws Exception {
        MessageDAO3 dao = new MessageDAO3(new SimpleConnectionPoolDataSource());
        ...
    }
}
```

提示>>> 应用程序经常通过 JNDI，从服务器上取得设定好的 DataSource，再从 DataSource 取得 Connection，将来读者接触到 Servlet/JSP 或其他 Java EE 应用领域，就会看到相关设定方式。

16.2.2 使用 ResultSet 卷动、更新数据

在 ResultSet 时，默认可以使用 next()移动数据光标至下一笔数据，然后使用 getXXX()方法来取得数据。从 JDBC 2.0 开始，ResultSet 不仅可以使用 **previous()**、**first()**、**last()**等方法前后移动数据光标，还可以调用 updateXXX()、updateRow()等方法进行数据修改。

在使用 Connection 的 createStatement()或 prepareStatement()方法建立 Statement 或 PreparedStatement 实例时，可以指定结果集类型与并行方式：

```
createStatement(int resultSetType, int resultSetConcurrency)
prepareStatement(String sql,int resultSetType, int resultSetConcurrency)
```

结果集类型可以指定 3 种设定：

- ResultSet.TYPE_FORWARD_ONLY(默认)
- ResultSet.TYPE_SCROLL_INSENSITIVE
- ResultSet.TYPE_SCROLL_SENSITIVE

注意>>> SQLite 只支持 TYPE_FORWARD_ONLY，若采用其他的设定会抛出 SQLException，接下来是在 MySQL 中测试过可行的代码段。

指定为 TYPE_FORWARD_ONLY，ResultSet 就只能前进数据光标。指定 TYPE_SCROLL_INSENSITIVE 或 TYPE_SCROLL_SENSITIVE，则 ResultSet 可以前后移动数据光标。两者差别在于 TYPE_SCROLL_INSENSITIVE 设定下，取得的 ResultSet 不会反映数据库中的数据修改，而 TYPE_SCROLL_SENSITIVE 会反映数据库中的数据修改。

更新设定可以有两种指定：

- ResultSet.CONCUR_READ_ONLY(默认)
- ResultSet.CONCUR_UPDATABLE

指定为 CONCUR_READ_ONLY，则只能用 ResultSet 进行数据读取，无法进行更新。指定为 CONCUR_UPDATABLE，就可以使用 ResultSet 进行数据更新。

在使用 Connection 的 createStatement()或 prepareStatement()方法建立 Statement 或 PreparedStatement 实例时，若没有指定结果集类型与并行方式，默认就是 TYPE_FORWARD_ONLY 与 CONCUR_READ_ONLY。如果想前后移动数据光标并想使用 ResultSet 进行更新，以下是个 Statement 指定的例子：

```
Statement stmt = conn.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
```

以下是个 PreparedStatement 指定的例子：

```
PreparedStatement stmt = conn.prepareStatement(
                    "SELECT * FROM messages",
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
```

在数据光标移动的 API 上，可以使用 **absolute()**、**afterLast()**、**beforeFirst()**、**first()**、**last()**进行绝对位置移动，使用 **relative()**、**previous()**、**next()**进行相对位置移动。这些方法如果成功移动就会返回 true，也可以使用 **isAfterLast()**、**isBeforeFirst()**、**isFirst()**、**isLast()** 判断目前位置。以下是个简单的程序范例片段：

```
Statement stmt = conn.createStatement("SELECT * FROM messages",
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery();
rs.absolute(2);                    // 移至第 2 列
rs.next();                         // 移至第 3 列
rs.first();                        // 移至第 1 列
boolean b1 = rs.isFirst();         // b1 是 true
```

如果要使用 ResultSet 进行数据修改，则有些条件限制：

- 必须选取单一表格。
- 必须选取主键。
- 必须选取所有 NOT NULL 的值。

在取得 ResultSet 之后要进行数据更新，必须移动至要更新的列(Row)，调用 **updateXXX()**方法(XXX 是类型)，然后调用 **updateRow()**方法完成更新。如果调用 **cancelRowUpdates()**可取消更新，但必须在调用 updateRow()前进行更新的取消。以下是一个使用 ResultSet 更新数据的例子：

```
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM messages",
                         ResultSet.TYPE_SCROLL_INSENSITIVE,
                         ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery();
rs.next();
rs.updateString(3, "caterpillar@openhome.cc");
rs.updateRow();
```

如果取得 ResultSet 后想直接进行数据的新增，则要先调用 **moveToInsertRow()**，之后调用 **updateXXX()**设定要新增的数据各个字段，然后调用 **insertRow()**新增数据。以下是一个使用 ResultSet 新增数据的例子：

```
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM messages",
                         ResultSet.TYPE_SCROLL_INSENSITIVE,
                         ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery();
rs.moveToInsertRow();
rs.updateString(2, "momor");
rs.updateString(3, "momor@openhome.cc");
rs.updateString(4, "blah..blah");
rs.insertRow();
rs.moveToCurrentRow();
```

如果取得 ResultSet 后想直接进行数据的删除，则要移动数据光标至想删除的列，调用 **deleteRow()**删除数据列。以下是一个使用 ResultSet 删除数据的例子：

```
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM messages",
                         ResultSet.TYPE_SCROLL_INSENSITIVE,
                         ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery();

rs.absolute(3);
rs.deleteRow();
```

16.2.3 批次更新

如果必须对数据库进行大量数据更新，单纯使用类似以下的程序片段并不适当：

```
Statement stmt = conn.createStatement();
while(someCondition) {
    stmt.executeUpdate(
      "INSERT INTO messages(name,email,msg) VALUES('…','…','…')");
}
```

每一次执行 executeUpdate()，其实都会向数据库发送一次 SQL，如果大量更新的 SQL 有一万笔，就等于通过网络进行了一万次的信息传送，网络传送信息实际上必须打开 I/O、进行路由等动作，如此进行大量更新，效能上其实不好。

可以使用 **addBatch()** 方法来收集 SQL，并使用 **executeBatch()** 方法将所收集的 SQL 传送出去。例如：

```
Statement stmt = conn.createStatement();
while(someCondition) {
    stmt.addBatch(
      "INSERT INTO messages(name,email,msg) VALUES('…','…','…')");
}
stmt.executeBatch();
```

提示»» 若是 sqlite-jdbc 驱动程序，其 Statement 实现的 addBatch() 使用了 Object[] 来收集 SQL，然而 executeBatch() 是使用 for 循环逐一取得 SQL 语句，自行控制 sqlite3_step()、sqlite3_perpare()、sqlite3_finalize() 的执行，而不是像 executeUpdate() 直接调用 sqlite3_exec()。

若是 MySQL 驱动程序的 Statement 操作，其 addBatch() 使用了 ArrayList 来收集 SQL，所有收集的 SQL，最后会串为一句 SQL，然后传送给数据库，也就是说假设大量更新的 SQL 有一万笔，这一万笔 SQL 会连接为一句 SQL，再通过一次网络传送给数据库，节省了 I/O、网络路由等动作所耗费的时间。

既然是使用批次更新，顾名思义，就是仅用在更新操作，所以批次更新的限制是，SQL 不能是 SELECT，否则会抛出异常。

使用 executeBatch() 时，SQL 的执行顺序就是 addBatch() 时的顺序。executeBatch() 会返回 int[]，代表每笔 SQL 造成的数据异动列数。在执行 executeBatch() 时，前面已打开的 ResultSet 会被关闭，执行过后收集 SQL 用的 List 会被清空，任何的 SQL 错误，会抛出 **BatchUpdateException**，可以使用这个对象的 **getUpdateCounts()** 取得 int[]，代表前面执行成功的 SQL 所造成的异动笔数。

前面举的是 Statement 的例子，如果是 PreparedStatement 要使用批次更新，可以使用 addBatch() 收集占位字符真正的数值，以下是个范例：

```
PreparedStatement stmt = conn.prepareStatement(
      "INSERT INTO messages(name,email,msg) VALUES(?, ?, ?)");
while(someCondition) {
    stmt.setString(1, "...");
    stmt.setString(2, "...");
    stmt.setString(3, "...");
    stmt.addBatch();        // 收集参数
}

stmt.executeBatch();        // 送出所有参数
```

提示>>> 除了在 API 上使用 addBatch()、executeBatch()等方法以进行批次更新之外，通常也会搭配关闭自动提交(auto commit)，在效能上也会有所影响，这在稍后说明交易时就会提到。

驱动程序本身是否支持批次更新也要注意一下。以 MySQL 为例，要支持批次更新，必须在 JDBC URL 上附加 rewriteBatchedStatements=true 参数才有实际的作用。

16.2.4 Blob 与 Clob

如果要将文档写入数据库，可以在数据库表格字段上使用 BLOB 或 CLOB 数据类型。BLOB 全名 Binary Large Object，用于存储大量的二进制数据，如图片文件、影音文件等；CLOB 全名 Character Large Object，用于存储大量的文字数据。

在 JDBC 中提供了 **java.sql.Blob** 与 **java.sql.Clob** 两个类分别代表 BLOB 与 CLOB 数据。以 Blob 为例，在写入数据时，可以通过 PreparedStatement 的 setBlob()来设定 Blob 对象；在读取数据时，可以通过 ResultSet 的 getBlob()取得 Blob 对象。

Blob 拥有 **getBinaryStream()**、**getBytes()**等方法，可以取得代表字段来源的 InputStream 或字段的 byte[]数据。Blob 拥有 **getCharacterStream()**、**getAsciiStream()**等方法，可以取得 Reader 或 InputStream 等数据。可以查看 API 文件来获得更详细的信息。

BLOB 字段也可以对应 byte[]或输入/输出串流。在写入数据时，可以使用 PreparedStatement 的 **setBytes()**来设定要存入的 byte[]数据，使用 **setBinaryStream()**来设定代表输入来源的 InputStream。在读取数据时，可以使用 ResultSet 的 **getBytes()**以 byte[]取得字段中存储的数据，或以 **getBinaryStream()**取得代表字段来源的 InputStream。

以下是取得代表文档来源的 InputStream 后，进行数据库存储的片段。

```
InputStream in = readFileAsInputStream("...");
PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO IMAGES(src, img) VALUE(?, ?)");
stmt.setString(1, "…");

stmt.setBinaryStream(2, in);

stmt.executeUpdate();
```

以下是取得代表字段数据源的 InputStream 片段：

```
PreparedStatement stmt = conn.prepareStatement(
    "SELECT img FROM IMAGES");
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    InputStream in = rs.getBinaryStream(1);
    //使用 InputStream 做数据读取
}
```

16.2.5 交易简介

交易的 4 个基本要求是原子性(Atomicity)、一致性(Consistency)、隔离行为(Isolation Behavior)与持续性(Durability)，以英文字母首字母简称为 ACID。

- 原子性：一个交易是一个单元工作(Unit of Work)，当中可能包括数个步骤，这些步骤必须全部执行成功。若有一个失败，则整个交易声明失败，交易中其他步骤必须撤销曾经执行过的动作，回到交易前的状态。

  在数据库上执行单元工作为数据库交易(Database Transaction)，单元中每个步骤就是每一句 SQL 的执行，要定义开始一个交易边界(通常是以一个 BEGIN 的指令开始)，所有 SQL 语句下达之后，COMMIT 确认所有操作变更，此时交易成功，或者因为某个 SQL 错误，ROLLBACK 进行撤销动作，此时交易失败。
- 一致性：交易作用的数据集合在交易前后必须一致。若交易成功，整个数据集合都必须是交易操作后的状态；若交易失败，整个数据集合必须与开始交易前一样没有变更，不能发生整个数据集合，部分有变更，部分没变更的状态。

  例如转账行为，数据集合涉及 A、B 两个账户，A 原有 20000，B 原有 10000，A 转 10000 给 B，交易成功的话，最后 A 必须变成 10000，B 变成 20000，交易失败的话，A 必须为 20000，B 为 10000，而不能发生 A 为 20000(未扣款)，B 也为 20000(已入款)的情况。
- 隔离性：在多人使用的环境下，每个用户可能进行自己的交易，交易与交易之间，必须互不干扰，用户不会意识到别的用户正在进行交易，就好像只有自己在进行操作一样。
- 持续性：交易一旦成功，所有变更必须保存下来，即使系统挂了，交易的结果也不能遗失，这通常需要系统软硬件架构的支持。

在原子性的要求上，在 JDBC 可以操作 `Connection` 的 **`setAutoCommit()`** 方法，给它 `false` 自变量，提示数据库开始交易，在下达一连串的 SQL 语句后，自行调用 `Connection` 的 **`commit()`**，提示数据库确认(Commit)操作。如果中间发生错误，则调用 **`rollback()`**，提示数据库撤销(Rollback)所有的执行。如下所示为一个示范流程：

```
Connection conn = null;
try {
    conn = dataSource.getConnection();

    conn.setAutoCommit(false);                    //取消自动提交

    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO …");
    stmt.executeUpdate("INSERT INTO …");

    conn.commit();                                //提交

}
catch(SQLException e) {
    e.printStackTrace();
    if(conn != null) {
        try {
            conn.rollback();                      //撤回
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
finally {
    ...
    if(conn != null) {
        try {
```

```
                conn.setAutoCommit(true);                    //回复自动提交
                conn.close();
            }
            catch(SQLException ex) {
                ex.printStackTrace();
            }
        }
    }
```

如果在交易管理时，仅想要撤回某个 SQL 执行点，则可以设定存储点(Save point)。例如：

```
Savepoint point = null;
try {
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO …");
    ...
    point = conn.setSavepoint();                    // 设定存储点
    stmt.executeUpdate("INSERT INTO …");
    ...
    conn.commit();
}
catch(SQLException e) {
    e.printStackTrace();
    if(conn != null) {
        try {
            if(point == null) {
                conn.rollback();
            }
            else {
                conn.rollback(point);               // 撤回存储点
                conn.releaseSavepoint(point);       // 释放存储点
            }
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
finally {
    ...
    if(conn != null) {
        try {
            conn.setAutoCommit(true);
            conn.close();
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

在批次更新时，不用每一笔都确认的话，也可以搭配交易管理。例如：

```
try {
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    while(someCondition) {
        stmt.addBatch("INSERT INTO …");
    }
    stmt.executeBatch();
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
            conn.commit();
        } catch(SQLException ex) {
            ex.printStackTrace();
            if(conn != null) {
                try {
                    conn.rollback();
                } catch(SQLException e) {
                    e.printStackTrace();
                }
            }
        } finally {
            ...
            if(conn != null) {
                try {
                    conn.setAutoCommit(true);
                    conn.close();
                }
                catch(SQLException ex) {
                    ex.printStackTrace();
                }
            }
        }
```

至于在隔离行为的支持上，JDBC 可以通过 Connection 的 **getTransactionIsolation()** 取得数据库目前的隔离行为设定，通过 **setTransactionIsolation()** 可提示数据库设定指定的隔离行为。可设定常数是定义在 Connection 上，如下所示：

- TRANSACTION_NONE
- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE

其中，TRANSACTION_NONE 表示对交易不设定隔离行为，仅适用于没有交易功能、以只读功能为主、不会发生同时修改字段的数据库。有交易功能的数据库，可能不理会 TRANSACTION_NONE 的设定提示。

要了解其他隔离行为设定的影响，首先要了解多个交易并行时，可能引发的数据不一致问题有哪些。以下逐一举例说明。

1. 更新遗失(Lost Update)

基本上就是指某个交易对字段进行更新的信息，因另一个交易的介入而遗失更新效力。举例来说，若某个字段数据原为 ZZZ，用户 A、B 分别在不同的时间点对同一字段进行更新交易，如图 16.12 所示。

单就用户 A 的交易而言，最后字段应该是 OOO，单就用户 B 的交易而言，最后字段应该是 ZZZ。在完全没有隔离两者交易的情况下，由于用户 B 撤销操作时间在用户 A 确认之后，最后字段结果会是 ZZZ，用户 A 看不到他更新确认的 OOO 结果，用户 A 发生更新遗失问题。

提示 »» 可想象有两个用户，若 A 用户打开文件之后，后续又允许 B 用户打开文件，一开始 A、B 用户看到的文件都有 ZZZ 文字，A 修改 ZZZ 为 OOO 后存储，B 修改 ZZZ 为 XXX 后又还原为 ZZZ 并存储，最后文件就为 ZZZ，A 用户的更新遗失。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

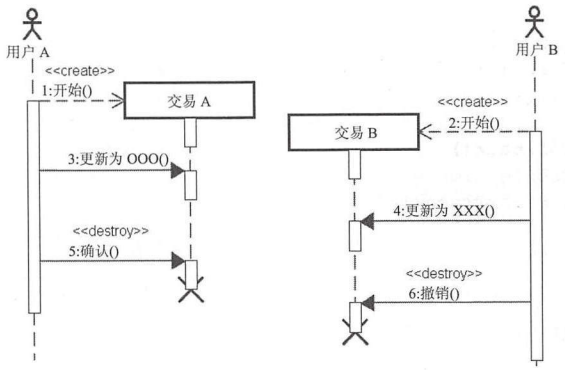


图 16.12　更新遗失

如果要避免更新遗失问题，可以设定隔离层级为“可读取未确认”(Read uncommitted)，也就是 A 交易已更新但未确认的数据，B 交易仅可做读取动作，但不可做更新的动作。JDBC 可通过 `Connection` 的 `setTransactionIsolation()` 设定为 `TRANSACTION_UNCOMMITTED` 来提示数据库确定此隔离行为。

数据库对此隔离行为的基本做法是，A 交易在更新但未确认，延后 B 交易的更新需求至 A 交易确认之后。以上例而言，交易顺序结果会变成如图 16.13 所示。

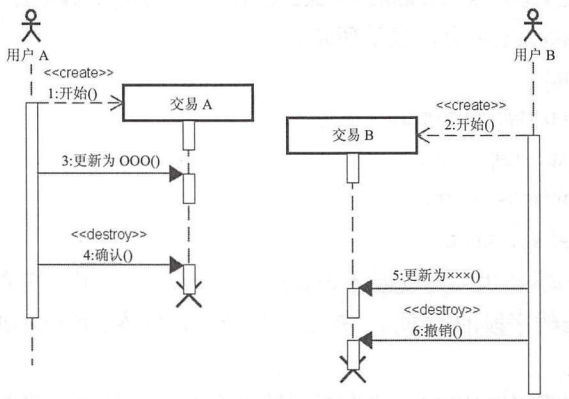


图 16.13　“可读取未确认”(Read uncommitted)避免更新遗失

提示 »» 可想象有两个用户，若 A 用户打开文件之后，后续只允许 B 用户以只读方式打开文件，B 用户若要能够写入，至少得等 A 用户修改完成关闭文档后。

提示数据库“可读取未确认”(Read uncommitted)的隔离层次之后，数据库至少得保证交易要避免更新遗失问题，通常这也是具备交易功能的数据库引擎会采取的最低隔离层级。不过这个隔离层级读取错误数据的概率太高，一般不会采用这种隔离层级。

2. 脏读(Dirty Read)

两个交易同时进行，其中一个交易更新数据但未确认，另一个交易就读取数据，就有可能发生脏读问题，也就是读到所谓脏数据(Dirty Data)，即不干净、不正确的数据，如图 16.14 所示。

用户 B 在 A 交易撤销前读取了字段数据为 OOO，如果 A 交易撤销了交易，那用户 B 读取的数据就是不正确的。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

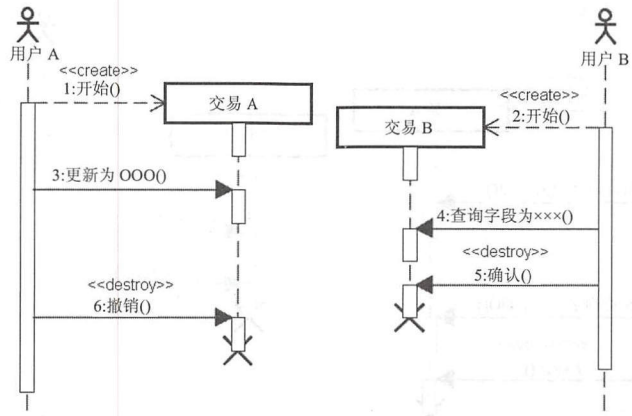


图 16.14　脏读

提示 »» 可想象有两个用户，若 A 用户打开文件并仍在修改期间，B 用户打开文件所读到的数据，就有可能是不正确的。

如果要避免脏读问题，可以设定隔离层级为“可读取确认”(Read Commited)，也就是交易读取的数据必须是其他交易已确认的数据。JDBC 可通过 `Connection` 的 `setTransaction Isolation()` 设定为 **`TRANSACTION_COMMITTED`** 来提示数据库确定此隔离行为。

数据库对此隔离行为的基本做法之一是，读取的交易不会阻止其他交易，未确认的更新交易会阻止其他交易。若是这个做法，交易顺序结果会变成如图 16.15 所示(若原字段为 ZZZ)。

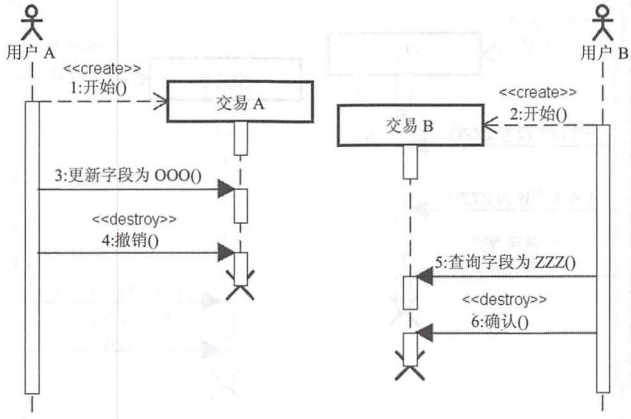


图 16.15　“可读取确认”避免脏读

提示 »» 可想象有两个用户，若 A 用户打开文件并仍在修改期间，B 用户就不能打开文件。但在数据库上这个做法影响效能较大，另一个基本做法是交易正在更新但尚未确定前先操作暂存表格，其他交易就不至于读取到不正确的数据。JDBC 隔离层级的设定提示，实际在数据库上如何操作，主要得以各家数据库在效能的考虑而定。

提示数据库“可读取确认”的隔离层次之后，数据库至少得保证交易要避免脏读与更新遗失问题。

3. 无法重复的读取(Unrepeatable Read)

某个交易两次读取同一字段的数据并不一致。例如，交易 A 在交易 B 更新前后进行数据的读取，则 A 交易会得到不同的结果，如图 16.16 所示(若原字段为 ZZZ)。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

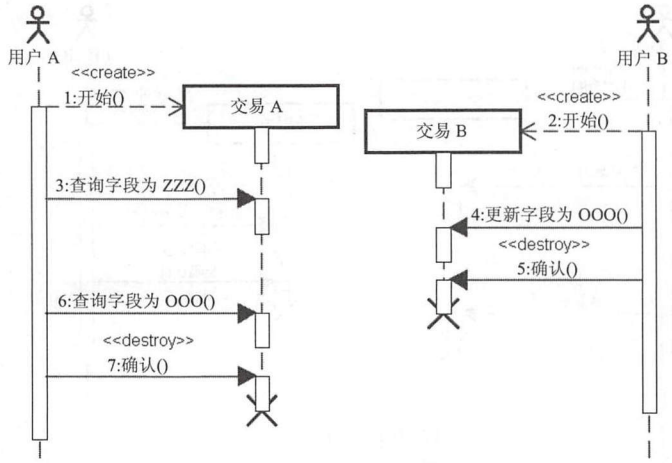


图 16.16　无法重复的读取

如果要避免无法重复的读取问题，可以设定隔离层级为“可重复读取”(Repeatable Read)，也就是同一交易内两次读取的数据必须相同。JDBC 可通过 `Connection` 的 `setTransactionIsolation()` 设定为 **`TRANSACTION_REPEATABLE_READ`** 来提示数据库确定此隔离行为。

数据库对此隔离行为的基本做法之一是，读取交易在确认前不阻止其他读取交易，但会阻止其他更新交易。若是这个做法，交易顺序结果会变成如图 16.17 所示(若原字段为 ZZZ)。

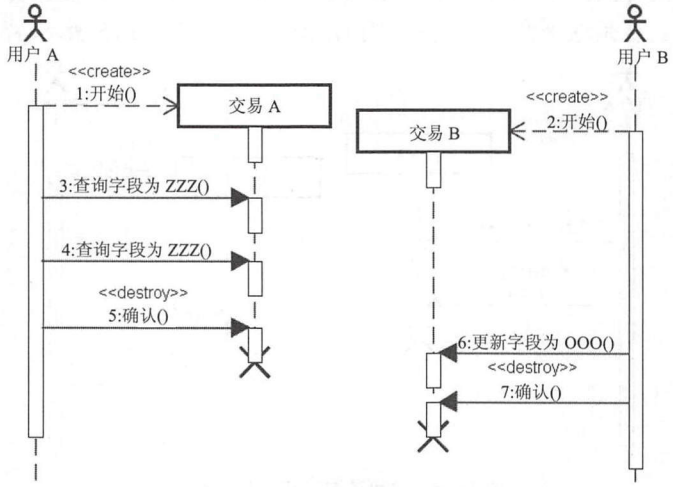


图 16.17　可重复读取

提示 »» 在数据库上这个做法影响效能较大，另一个基本做法是交易正在读取但尚未确认前，另一交易会在暂存表格上更新。

提示数据库“可重复读取”的隔离层次之后，数据库至少得保证交易要避免无法重复读取、脏读与更新遗失问题。

4. 幻读(Phantom Read)

同一交易期间，读取到的数据笔数不一致。例如，交易 A 第一次读取得到 5 笔数据，此时交易 B 新增了 1 笔数据，导致交易 B 再次读取得到 6 笔数据。

如果隔离行为设定为可重复读取，但发生幻读现象，可以设定隔离层级为“可循序”

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

(Serializable)，也就是在有交易时若有数据不一致的疑虑，交易必须可以照顺序逐一进行。JDBC可通过 Connection 的 setTransactionIsolation() 设定为 **TRANSACTION_SERIALIZABLE** 来提示数据库确定此隔离行为。

提示»» 交易若真的一个一个循序进行，对数据库的效率影响会很大，实际也许未必直接阻止其他交易或真的循序进行，例如采取暂存表格方式。事实上，只要能符合 4 个交易隔离要求，各家数据库会寻求最有效率的解决方式。

表 16.2 整理了各个隔离行为可预防的问题。

表 16.2 隔离行为与可预防的问题

隔离行为	更新遗失	脏读	无法重复的读取	幻读
可读取未确认	预防			
可读取确认	预防	预防		
可重复读取	预防	预防	预防	
可循序	预防	预防	预防	预防

如果想通过 JDBC 得知数据库是否支持某个隔离行为设定，可以通过 Connection 的 **getMetaData()** 取得 **DatabaseMetadata** 对象，通过 DatabaseMetadata 的 **supportsTransactionIsolationLevel()** 得知是否支持某个隔离行为。例如：

```
DatabaseMetadata meta = conn.getMetaData();
boolean isSupported = meta.supportsTransactionIsolationLevel(
        Connection.TRANSACTION_READ_COMMITTED);
```

16.2.6 metadata 简介

metadata 即“诠读数据的数据”(Data About Data)。例如，数据库是用来存储数据的地方，然而数据库本身产品名称为何？数据库中有几个数据表格？表格名称为何？表格中有几个字段等？这些信息就是所谓的 metadata。

在 JDBC 中，可以通过 **Connection** 的 **getMetaData()** 方法取得 **DatabaseMetaData** 对象，通过这个对象提供的各个方法，可以取得数据库整体信息。ResultSet 表示查询到的数据，数据本身的字段、类型等信息可以通过 **ResultSet** 的 **getMetaData()** 方法，取得 **ResultSetMetaData** 对象，通过这个对象提供的相关方法，就可以取得域名、字段类型等信息。

提示»» DatabaseMetaData 或 ResultSetMetaData 本身 API 使用上不难，问题点在于各家数据库对某些名词的定义不同，必须查阅数据库厂商手册搭配对应的 API，才可以取得想要的信息。

以下举个例子，利用 JDBC 的 metadata 相关 API，取得前面文档管理范例 messages 表格相关信息：

JDBCDemo MessagesInfo.java

```
package cc.openhome;

import java.sql.*;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
import java.util.*;
import javax.sql.DataSource;

public class MessagesInfo {
    private DataSource dataSource;

    public MessagesInfo(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public List<ColumnInfo> getAllColumnInfo() {
        List<ColumnInfo> infos = null;
        try(Connection conn = dataSource.getConnection()) { ❶查询messages表格所有字段
            DatabaseMetaData meta = conn.getMetaData();
            ResultSet crs = meta.getColumns("demo.sqlite", null, "messages", null);
            infos =  new ArrayList<>(); ←— ❷用来收集字段信息
            while(crs.next()) {
                ColumnInfo info = toColumnInfo(crs); ←— ❸封装域名、类型、大小、可否
                infos.add(info);                      为空、默认值等信息
            }
        }
        catch(SQLException ex) {
            throw new RuntimeException(ex);
        }
        return infos;
    }

    private ColumnInfo toColumnInfo(ResultSet crs) throws SQLException {
        ColumnInfo info = new ColumnInfo();
        info.setName(crs.getString("COLUMN_NAME"));
        info.setType(crs.getString("TYPE_NAME"));
        info.setSize(crs.getInt("COLUMN_SIZE"));
        info.setNullable(crs.getBoolean("IS_NULLABLE"));
        info.setDef(crs.getString("COLUMN_DEF"));
        return info;
    }
}
```

在调用 `getAllColumnInfo()`时，会先从 `Connection` 上取得 `DatabaseMetaData`，以查询数据库中指定表格的字段❶，这会取得一个 `ResultSet`。接着从 `ResultSet` 上逐一取得各个想要的信息，封装为 `ColumnInfo` 对象❸，并收集在 `List` 中返回❷。

`ColumnInfo` 只是自定义的简单类，用来封装字段各个信息。如：

JDBCDemo ColumnInfo.java

```
package cc.openhome;

import java.io.Serializable;

public class ColumnInfo implements Serializable {
    private String name;
    private String type;
    private int size;
    private boolean nullable;
    private String def;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public int getSize() {
        return size;
    }
    public void setSize(int size) {
        this.size = size;
    }
    public boolean isNullable() {
        return nullable;
    }
    public void setNullable(boolean nullable) {
        this.nullable = nullable;
    }
    public String getDef() {
        return def;
    }
    public void setDef(String def) {
        this.def = def;
    }
}
```

可以使用以下范例来运用 MessagesInfo 取得字段信息。

JDBCDemo MessagesInfoDemo.java

```
package cc.openhome;

import java.io.IOException;
import static java.lang.System.out;

public class MessagesInfoDemo {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        MessagesInfo messagesInfo =
                new MessagesInfo (new SimpleConnectionPoolDataSource());
        out.println("名称\t型态\t为空\t默认");
        MessagesInfo.getAllColumnInfo().forEach(info -> {
            out.printf("%s\t%s\t%s\t%s%n",
                    info.getName(),
                    info.getType(),
                    info.isNullable(),
                    info.getDef());
        });
    }
}
```

❶ 传入 DataSource 实例

❷ 显示字段信息

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

执行结果如下所示：

```
名称        类型          为空          默认
id          INTEGER       false         null
name        TEXT          false         null
email       TEXT          false         null
msg TEXT    false         null
```

16.2.7 RowSet 简介

JDBC 定义了 **javax.sql.RowSet** 接口，用以代表数据的列集合。这里的数据并不一定是数据库中的数据，可以是电子表格数据、XML 数据或任何具有列集合概念的数据源。

RowSet 是 ResultSet 的子接口，所以具有 ResultSet 的行为，可以使用 RowSet 对列集合进行增删查改。RowSet 也新增了一些行为，如通过 setCommand()设定查询指令、通过 execute()执行查询指令以填充数据等。

提示»» 在 Oracle 的 JDK 中附有 RowSet 的非标准操作，其包名称是 com.sun.rowset。

RowSet 定义了列集合基本行为，其下有 **JdbcRowSet**、**CachedRowSet**、**FilteredRowSet**、**JoinRowSet** 与 **WebRowSet** 5 个标准列集合子接口，定义在 **javax.sql.rowset** 包中。其继承关系如图 16.18 所示。

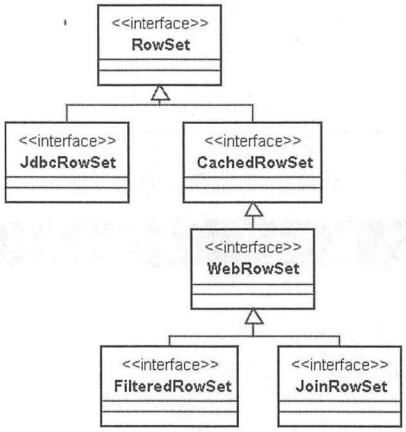


图 16.18 RowSet 接口继承架构

由于 javax.sql.rowset 包在 JDK9 中是放在 java.sql.rowset 模块之中，若采用模块化设计，记得在模块描述文件加上 requires java.sql.rowset。

JdbcRowSet 是联机式(Connected)的 RowSet，也就是操作 JdbcRowSet 期间，会保持与数据库的联机，可视为取得、操作 ResultSet 的行为封装，可简化 JDBC 程序的撰写，或作为 JavaBean 使用。

CachedRowSet 则为脱机式(Disconnected)的 RowSet(其子接口当然也是)，在查询并填充完数据后，就会断开与数据源的联机，而不用占据相关联机资源，必要时也可以再与数据源联机进行数据同步。

以下先以 JdbcRowSet 为例，介绍 RowSet 的基本操作。在这里使用的成果是 Oracle/Sun JDK 附带的 JdbcRowSetImpl。在 JDK6 之前，可以这样建立 JdbcRowSet 实例：

```
JdbcRowSet rowset = new JdbcRowSetImpl();
```

在JDK7之后，新增了 **javax.sql.rowset.RowSetFactory** 接口与 **javax.sql.rowset.RowSetProvider** 类，可以使用 **RowSetProvider.newFactory()** 取得 RowSetFactory 操作对象，再利用 RowSetFactory 的 **createJdbcRowSet()**、**createCachedRowSet()** 等方法，建立 RowSet 实例。例如建立 JdbcRowSet 实例：

```
RowSetFactory rowSetFactory = RowSetProvider.newFactory();
JdbcRowSet rowset = rowSetFactory.createJdbcRowSet();
```

如果使用 Oracle/Sun JDK，以上程序片段会取得 JdbcRowSetImpl 实例，若有其他厂商成果，可以在启动 JVM 时，利用系统属性 javax.sql.rowset.RowSetFactory 指定，例如-Djavax.sql.rowset.RowSetFactory=com.other.rowset.RowSetFactoryImpl。

提示»» 由于 SQLite 在建立 Statement 或 PreparedStatement 时只支持 TYPE_FORWARD_ONLY，若采用其他设定会抛出 SQLException，而 Oracle JDK 附带的 JdbcRowSetImpl 会采用 TYPE_SCROLL_INSENSITIVE，因此无法配合 SQLite 使用，接下来提供的是在 MySQL 中测试过的可行代码段。

要使用 RowSet 查询数据，基本上可以这样：

```
rowset.setUrl("jdbc:mysql://localhost:3306/demo");
rowset.setUsername("root");
rowset.setPassword("openhome");
rowset.setCommand("SELECT * FROM t_messages WHERE id = ?");
rowset.setInt(1, 1);
rowset.execute();
```

使用 **setUrl()** 设定 JDBC URL，使用 **setUsername()** 设定用户名称，使用 **setPassword()** 设定密码，使用 **setCommand()** 设定查询 SQL。JdbcRowSet 也有 **setAutoCommit()** 与 **commit()** 方法，可以进行交易控制。

由于 RowSet 是 ResultSet 的子接口，要取得各字段数据，只要如 ResultSet 操作即可，若要使用 RowSet 进行增删改的动作，也是与 ResultSet 相同。

如果在查询之后，想要脱机进行操作，则可以使用 CachedRowSet 或其子接口操作对象，查询数据之后可以直接使用 **close()** 关闭联机。若在相关更新操作之后，想再与数据源进行同步，则可以调用 **acceptChanges()** 方法。例如：

```
conn.setAutoCommit(false);          // conn 是 Connection
rowSet.acceptChanges(conn);         // rowSet 是 CachedRowSet
conn.setAutoCommit(true);
```

WebRowSet 是 CachedRowSet 的子接口，其不仅具备脱机操作，还能进行 XML 读写。例如以下的 TMessageUtil.writeXml() 方法，可以读取数据库的表格数据，然后对指定的 OutputStream 写出 XML：

```
public static void writeXml(OutputStream outputStream) throws Exception {
    try(WebRowSet rowset = RowSetProvider.newFactory().createWebRowSet()) {
        rowset.setUrl("jdbc:mysql://localhost:3306/demo");
        rowset.setUsername("root");
        rowset.setPassword("openhome");
        rowset.setCommand("SELECT * FROM messages");
        rowset.execute();
        rowset.writeXml(outputStream);
    }
}
```


非卖品，仅供非商业用途或交流学习使用

从JDK7之后，RowSet都是java.lang.AutoCloseable的子接口，可以使用尝试自动关闭资源语法。范例中使用WebRowSet中的writeXML()，可以将WebRowSet的metadata、属性与数据以XML格式写出。执行结果如下：

```
...
  <data>
    <currentRow>
      <columnValue>1</columnValue>
      <columnValue>良葛格</columnValue>
      <columnValue>caterpillar@openhome.cc</columnValue>
      <columnValue>这是一篇测试留言！</columnValue>
    </currentRow>
    <currentRow>
      <columnValue>2</columnValue>
      <columnValue>毛美眉</columnValue>
      <columnValue>momor@openhome.cc</columnValue>
      <columnValue>我来留言啰！</columnValue>
    </currentRow>
  </data>
</webRowSet>
```

只要TMessageUtil.writeXml()指定的目的地，有办法处理XML，就可以自行组织出想要的信息或画面。

FilteredRowSet可以对列集合进行过滤，实现类似SQL中WHERE等条件式的功能。通过**setFilter()**方法，指定操作**javax.sql.rowset.Predicate**的对象。其定义如下：

```
boolean evaluate(Object value, int column)
boolean evaluate(Object value, String columnName)
boolean evaluate(RowSet rs)
```

Predicate的evaluate()方法返回true，表示该列要包括在过滤后的列集合中。

JoinRowSet则可以让你结合两个RowSet对象，实现类似SQL中JOIN的功能。可以通过**setMatchColumn()**指定要结合的行，然后使用**addRowSet()**来加入RowSet进行结合。例如：

```
rs1.setMatchColumn(1);
rs2.setMatchColumn(2);
JoinRowSet jrs = new JoinRowSetImpl();
jrs.addRowSet(rs1);
jrs.addRowSet(rs2);
```

在这个范例片段执行过后，JoinRowSet中就会是原本两个RowSet结合的结果。也可以通过**setJoinType()**指定结合的方式，可指定的常数定义在JoinRowSet中，包括CROSS_JOIN、FULL_JOIN、INNER_JOIN、LEFT_OUTER_JOIN与RIGHT_OUTER_JOIN。

提示>>> API文件对RowSet的文件说明清晰明了，更多有关RowSet或JDBC的说明，读者也可以参考*JDBC Basic*，网址如下：
http://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html

16.3 重点复习

JDBC(Java DataBase Connectivity)是用于执行SQL的解决方案，开发人员使用JDBC的标

非卖品，仅供非商业用途或交流学习使用

准接口，数据库厂商则对接口进行操作，开发人员无须接触底层数据库驱动程序的差异性。

厂商在操作 JDBC 驱动程序时，按照方式可将驱动程序分为 4 种类型：

- Type 1 JDBC-ODBC Bridge Driver
- Type 2 Native API Driver
- Type 3 JDBC-Net Driver
- Type 4 Native Protocol Driver

数据库操作相关的 JDBC 接口或类都位于 `java.sql` 包中。要连接数据库，可以向 `DriverManager` 取得 `Connection` 对象。`Connection` 是数据库联机的代表对象，一个 `Connection` 对象就代表一个数据库联机。`SQLException` 是在处理 JDBC 时经常遇到的一个异常对象，为数据库操作过程发生错误时的代表对象。

明确定义的模块称为显式模块，若是在类路径上包含 JAR 文件，会使得 JAR 文件中的类被归类在未命名模块，显式模块无法 `requires` 未命名模块。

若是采用模块化设计，可以将未支持模块化设计的 JAR 文件放在模块路径(使用 `--module-path` 指定)，这会使得该 JAR 文件被视为自动模块，自动模块也是命名模块的一种。

基于兼容性，自动模块有隐含的模块定义，可以读取其他模块，其他模块也可以存取(以及深层反射)自动模块。应用程序在迁移至模块化设计的过程中，自动模块会是未命名模块至显式模块之间的桥梁。

取得联机等与数据库来源相关的行为规范在 `javax.sql.DataSource` 接口，实际如何取得 `Connection` 则由操作接口的对象来负责。

`Connection` 是数据库连接的代表对象，接下来要执行 SQL 的话，必须取得 `java.sql.Statement` 对象，它是 SQL 描述的代表对象。可以使用 `Connection` 的 `createStatement()` 来建立 `Statement` 对象。

`Statement` 的 `executeQuery()` 方法则是用于 SELECT 等查询数据库的 SQL，`executeUpdate()` 会返回 `int` 结果，表示数据变动的笔数，`executeQuery()` 会返回 `java.sql.ResultSet` 对象，代表查询的结果，查询的结果会是一笔一笔的数据。可以使用 `ResultSet` 的 `next()` 来移动至下一笔数据，它会返回 `true` 或 `false` 表示是否有下一笔数据，接着可以使用 `getXXX()` 来取得数据。

在使用 `Connection`、`Statement` 或 `ResultSet` 时，要将之关闭以释放相关资源。

如果有些操作只是 SQL 语句当中某些参数会有所不同，其余的 SQL 子句皆相同，则可以使用 `java.sql.PreparedStatement`。可以使用 `Connection` 的 `preparedStatement()` 方法建立好一个预先编译(Precompile)的 SQL 语句，当中参数会变动的部分，先指定“`?`”这个占位字符。等到需要真正指定参数执行时，再使用相对应的 `setInt()`、`setString()` 等方法，指定“`?`”处真正应该有的参数。

16.4 课后练习

请尝试撰写一个 `JdbcTemplate` 类封装 JDBC 更新操作，可以使用 `update()` 与 `queryForList()` 方法。如下：

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
```


非卖品，仅供非商业用途或交流学习使用

```
jdbcTemplate.update(
        "CREATE TABLE messages (" +
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL," +
            "name TEXT NOT NULL," +
            "email TEXT NOT NULL," +
            "msg TEXT NOT NULL)"
);


jdbcTemplate.update(
        "INSERT INTO messages(name, email, msg) VALUES (?,?,?)",
        "测试员", "tester@openhome.cc", "这是一个测试留言");


jdbcTemplate.queryForList("SELECT * FROM messages")
            .forEach(message -> {
                out.printf("%d\t%s\t%s\t%s%n",
                        message.get("id"),
                        message.get("name"),
                        message.get("email"),
                        message.get("msg"));
            });
```

其中 dataSource 参考至 DataSource 操作对象，update()第一个参数接受更新 SQL，之后的不定长度自变量可接受 SQL 中占位字符?实际数据，不定长度自变量部分不一定是字符串，也可接受表 16.1 列出的数据类型，而 queryForList()的返回值可以是 List<Map>。

提示 »» 搜寻关键词 JdbcTemplate 了解相关设计方式。

非卖品，仅供非商业用途或交流学习使用

Chapter 17 反射与类加载器

学习目标

- 取得.class 文档信息
- 动态生成对象与操作方法
- 认识模块与反射的权限设定
- 了解 JDK9 类加载器层级
- 使用 `ClassLoader` 实例

非卖品，仅供非商业用途或交流学习使用

17.1 运用反射

虽然拿到一个.class 文档对它一无所知，但文档本身记录了许多信息。Java 在需要使用到某个类时才会将.class 文档载入，在 JVM 产生 **java.lang.Class** 实例代表该文档，从 Class 实例开始，就可以获得类的许多类型。.class 文档反映了类基本信息，因而从 Class 等 API 取得类信息的方式就称为反射(Reflection)。

由于 JDK9 支持模块化，开发者在采取模块设计时，如何在不破坏模块封装下又能运用反射机制的弹性，就成了认识反射时必须知道的一大课题。就算你熟悉 Java，也对反射有一定的认识，我建议还是要仔细阅读本章。

在本章一开始，会先对 **java.base** 模块以及范例项目的 **cc.openhome** 模块下的类进行反射(没有设定 **SecurityManager** 的情况下)，17.1.7 节再来介绍不同模块间的反射必须注意的事项。

注意»» 有关 JDK8 或以前版本的反射，为了避免同时包含新旧机制的内容，造成理解上的混乱，本书中不再赘述，读者可以寻找《Java JDK 8 学习笔记》中有关反射的内容，或者参考在线文件：
openhome.cc/Gossip/JavaEssence/index.html#Reflection

17.1.1 Class 与.class 文档

Java 真正需要某个类时才会加载对应的.class 文档，而非在程序启动时就加载所有类。因为大多数用户只会使用应用程序部分资源，在需要某些功能时才加载对应资源，可以让系统资源运用更有效率(Java 本来就是为资源有限的小型设备而设计，这是必然的考虑)。

java.lang.Class 的实例代表 Java 应用程序运行时加载的.class 文档，类、接口、Enum 等编译过后，都会生成.class 文档，所以 Class 可用来包含类、接口、Enum 等信息。Class 类没有公开(public)构造函数，实例是由 JVM 自动产生，每个.class 文档加载时，JVM 会自动生成对应的 Class 对象。

可以通过 Object 的 **getClass()** 方法，或者通过 **.class** 常量(Class literal)取得每个对象对应的 Class 对象。如果是基本类型，也可以使用对应的打包类加上.TYPE 取得 Class 对象，例如 Integer.TYPE 可取得代表 int 的 Class 对象。

注意»» 使用 Integer.TYPE 可取得代表 int 基本类型的 Class，也可以使用 int.class 取得；如果要取得代表 Integer.class 文档的 Class，则必须使用 Integer.class。

在取得 Class 对象后，就可以操作 Class 对象的公开方法取得类基本信息。例如，以下是可取得 String 类的 Class 实例，并从中获得 String 的基本信息。

Reflection ClassInfo.java

```
package cc.openhome;

import static java.lang.System.out;

public class ClassInfo {
    public static void main(String[] args) {
```


非卖品，仅供非商业用途或交流学习使用

```
        Class clz = String.class;

        out.printf("类名称：%s%n", clz.getName());
        out.printf("是否为接口：%s%n", clz.isInterface());
        out.printf("是否为基本类型：%s%n", clz.isPrimitive());
        out.printf("是否为数组对象：%s%n", clz.isArray());
        out.printf("父类名称：%s%n", clz.getSuperclass().getName());
        out.printf("所在模块：%s%n", clz.getModule().getName());
    }
}
```

在 JDK9 中对于任何应用程序，都依赖在 java.base 模块，从 java.base 模块中的类开始认识反射的运用最为简单。在 JDK9 中 class 上也新增了 getModule()方法，可以让你取得代表模块的 java.lang.Module 实例，以便进一步取得有关模块的信息，19.2.1 节会进一步介绍如何使用。执行结果如下：

```
类名称：java.lang.String
是否为接口：false
是否为基本类型：false
是否为数组对象：false
父类名称：java.lang.Object
所在模块：java.base
```

Java 真正需要类时才会加载.class 文档，也就是要使用指定类生成对象时(或使用 Class.forName()、使用 java.lang.ClassLoader 实例的 loadClass()加载类时，稍后说明)。使用类声明参考名称并不会加载.class 文档(编译程序仅会检查对应的.class 文档是否存在)。例如，可设计测试类来印证：

Reflection Some.java

```
package cc.openhome;

public class Some {
    static {
        System.out.println("载入 Some.class 文档");
    }
}
```

Some 类定义了 static 区块，默认首次加载.class 文档时会执行静态区块(默认，是因为可以设定加载.class 文档时不执行 static 区块，稍后介绍)。通过在文本模式下显示类型，可以了解何时加载.class 文档。例如：

Reflection SomeDemo.java

```
package cc.openhome;

import static java.lang.System.out;

public class SomeDemo {
    public static void main(String[] args) {
        Some s;
        out.println("声明 Some 参考名称");
        s = new Some();
        out.println("生成 Some 实例");
    }
}
```


非卖品，仅供非商业用途或交流学习使用

声明 Some 参考名称不会载入 Some.class 文档，使用 new 生成对象时才会加载类(因为必须从.class 文档得知构造函数定义为何)，所以执行 new Some() 时，才会发现 static 区块执行类型。执行结果如下：

```
声明 Some 参考名称
载入 Some.class 文档
生成 Some 实例
```

类信息是在编译时期存储在.class 文档，这是 Java 支持执行运行时类型识别(RTTI，Run-Time Type Information 或 Run-Time Type Identification)的方式。编译时期若使用到相关类，编译程序会检查对应的.class 文档中记载的信息，以确定是否可完成编译。在执行时期使用某类时，会先检查是否有对应的 Class 对象，如果没有，会加载对应的.class 文档并生成对应的 Class 实例。

默认 JVM 只会用一个 Class 实例来代表一个.class 文档(确切的说法是，通过同一类加载器载入的.class 文档，只会有一个对应的 Class 实例)，每个类的实例都会知道自己由哪个 Class 实例生成。默认使用 getClass() 或.class 取得的 Class 实例会是同一个对象。例如：

```
jshell> "".getClass() == String.class;
$1 ==> true
```

17.1.2 使用 Class.forName()

在某些应用中，无法事先知道开发人员要使用哪个类。例如，事先不知道开发人员会使用哪个厂商的 JDBC 驱动程序，也就不知道厂商操作 java.sql.Driver 接口的类名称为何，因而必须让开发人员可以事后指定类名称来动态加载类。

使用 **Class.forName()** 方法实现动态加载类，可用字符串指定类名称来获得类相关信息。例如：

Reflection InfoAbout.java

```
package cc.openhome;

import static java.lang.System.out;

public class InfoAbout {
    public static void main(String[] args) {
        try {
            Class clz = Class.forName(args[0]);
            out.printf("类名称: %s%n", clz.getName());
            out.printf("是否为接口: %s%n", clz.isInterface());
            out.printf("是否为基本类型: %s%n", clz.isPrimitive());
            out.printf("是否为数组: %s%n", clz.isArray());
            out.printf("父类: %s%n", clz.getSuperclass().getName());
            out.printf("所在模块: %s%n", clz.getModule().getName());
        } catch (ArrayIndexOutOfBoundsException e) {
            out.println("没有指定类名称");
        } catch (ClassNotFoundException e) {
            out.printf("找不到指定的类 %s%n", args[0]);
        }
    }
}
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

Class.forName()方法在找不到指定类时会抛出 **ClassNotFoundException** 异常。如果启动JVM时指定的命令行自变量是 java.lang.String，则显示结果与上一个范例执行结果相同。

Class.forName()另一版本可以让指定类名称、加载类时是否执行静态区块与类加载器：

```
static Class forName(String name, boolean initialize, ClassLoader loader)
```

之前曾经说过，默认加载.class 文件时会执行类中定义的 static 区块。可以使用 forName()第二个版本，将 initialize 设定为 false，这样加载.class 文档时并不会立即执行 static 区块，而会在建立类实例时才执行 static 区块。例如：

Reflection SomeDemo2.java

```
package cc.openhome;

import static java.lang.System.out;

class Some2 {
    static {
        out.println("[执行静态区块]");
    }
}

public class SomeDemo2 {
    public static void main(String[] args) throws ClassNotFoundException {
        Class clz = Class.forName("cc.openhome.Some2", false,
                SomeDemo2.class.getClassLoader());
        out.println("已载入 Some2.class ");
        Some2 s;
        out.println("声明 Some 参考名称");
        s = new Some2();
        out.println("生成 Some 实例");
    }
}
```

由于在使用 Class.forName()方法时，设定 initialize 为 false，所以加载.class 文档时并不会执行静态区块，而会在使用类建立对象时才执行静态区块。第二个版本的 Class.forName()方法会需要一个类加载器，可取得代表 SomeDemo2.class 文档的 Class 实例后，通过 **getClassLoader()**方法，取得加载 SomeDemo2.class 文档的类加载器，再传递给 Class.forName()使用。执行结果如下：

```
已载入 Some2.class
声明 Some 参考名称
[执行静态区块]
生成 Some 实例
```

事实上，如果使用第一个版本的 Class.forName()方法，等同于：

```
Class.forName(className, true, currentLoader);
```

其中 currentLoader 是目前类的类加载器，也就是如果在 A 类中使用 Class.forName()第一个版本，默认就是用 A 类的类加载器来载入类。

严禁网络传播本PDF，违者责任自负！
非卖品，仅供非商业用途或交流学习使用

17.1.3 从 Class 获得信息

Class 对象代表加载的.class 文档，取得 Class 对象后，就可以取得.class 文档中记载的信息，如包、构造函数、方法成员、数据成员等类型。每个类型会有对应的类型，例如模块对应的类型为 **java.lang.Module**，包对应类型是 **java.lang.Package**，构造函数对应类型是 **java.lang.reflect.Constructor**，方法成员对应类型是 **java.lang.reflect.Method**，数据成员对应类型是 **java.lang.reflect.Field** 等。例如要取得指定 String 类的包名称，可以这样：

```
Package p = String.class.getPackage();
System.out.println(p.getName());      // 显示 java.lang
```

可以分别取回 Field、Constructor、Method 等对象，分别代表数据成员、构造函数与方法成员。以下范例是可取得类基本信息的程序：

Reflection ClassViewer.java

```
package cc.openhome;

import static java.lang.System.out;
import java.lang.reflect.*;

public class ClassViewer {
    public static void main(String[] args) {
        try {
            ClassViewer.view(args[0]);
        } catch (ArrayIndexOutOfBoundsException e) {
            out.println("没有指定类");
        } catch (ClassNotFoundException e) {
            out.println("找不到指定类");
        }
    }

    public static void view(String clzName) throws ClassNotFoundException {
        Class clz = Class.forName(clzName);

        showPackageInfo(clz);
        showClassInfo(clz);

        out.println("{");

        showFiledsInfo(clz);
        showConstructorsInfo(clz);
        showMethodsInfo(clz);

        out.println("}");
    }


 private static void showModuleName(Class clz) {
        Module m = clz.getModule();                    // 取得模块代表对象
        out.printf("module %s;%n", m.getName());
    }

    private static void showPackageName(Class clz) {
       int modifier = clz.getModifiers();              // 取得类型修饰常数
        out.printf("%s %s %s",
                Modifier.toString(modifier),           // 将常数转为字符串表示
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
                Modifier.isInterface(modifier) ? "interface" : "class",
                clz.getName() // 取得类名称
        );
    }

    private static void showFiledsInfo(Class clz) throws SecurityException {
        // 取得声明的数据成员代表对象
        Field[] fields = clz.getDeclaredFields();
        for (Field field : fields) {
            // 显示权限修饰，例如 public、protected、private
            out.printf("\t%s %s %s;%n",
                    Modifier.toString(field.getModifiers()),
                    field.getType().getName(),          // 显示类型名称
                    field.getName()                     // 显示数据成员名称
            );
        }
    }

    private static void showConstructorsInfo(Class clz) throws SecurityException {
        // 取得声明的创建方法代表对象
        Constructor[] constructors = clz.getDeclaredConstructors();
        for (Constructor constructor : constructors) {
            // 显示权限修饰，例如 public、protected、private
            out.printf("\t%s %s();%n",
                    Modifier.toString(constructor.getModifiers()),
                    constructor.getName()               // 显示构造函数名称
            );
        }
    }

    private static void showMethodsInfo(Class clz) throws SecurityException {
        // 取得声明的方法成员代表对象
        Method[] methods = clz.getDeclaredMethods();
        for (Method method : methods) {
            // 显示权限修饰，例如 public、protected、private
            out.printf("\t%s %s %s();%n",
                    Modifier.toString(method.getModifiers()),
                    method.getReturnType().getName(), // 显示返回值类型名称
                    method.getName()                  // 显示方法名称
            );
        }
    }
}
```

如果命令行自变量指定了 `java.lang.String`，则执行结果如下：

```
module java.base;
package java.lang;
public final class java.lang.String{
        private final [B value;
        private final byte coder;
        private int hash;
        private static final long serialVersionUID;
        static final boolean COMPACT_STRINGS;
        private static final [Ljava.io.ObjectStreamField; serialPersistentFields;
        public static final java.util.Comparator CASE_INSENSITIVE_ORDER;
        static final byte LATIN1;
        static final byte UTF16;
        public java.lang.String();
```


严禁网络传播本PDF，违者责任自负！

```
        ...
        public int hashCode();
        public void getChars();
        public volatile int compareTo();
        public int compareTo();
        ...
    }
```

17.1.4 从 Class 建立对象

如果知道类名称，可以使用 new 关键字建立实例。如果事先不知道类名称呢？可以利用 Class.forName()动态加载.class 文档，取得 Class 对象之后，调用 Class 的 **getConstructor()**或 **getDeclaredConstructor()**方法，取得代表构造函数的 Constructor 对象，利用其 newInstance()方法建立类实例。例如：

```
Class clz = Class.forName(args[0]);
Object obj = clz.getDeclaredConstructor().newInstance();
```

注意»»» Class 的 newInstance()方法，在 JDK9 时被标示为废弃了，因为若构造函数声明抛出了非受检异常，调用此方法等同于略过编译时期检查；Constructor 的 newInstance()方法会抛出受检的 InvocationTargetException，而任何构造函数抛出的异常都会被包裹在 InvocationTargetException 实例之中。

如果实际加载类定义了无参数构造函数，就可以使用这种方式创建对象，为何会有事先不知道类名称，又要建立类实例的需求？例如，想采用影片链接库来播放动画，然而负责操作影片链接库的部门迟迟还没动工，怎么办呢？可以利用接口定义出影片链接库该有的功能。例如：

Reflection Player.java

```
package cc.openhome;

public interface Player {
    void play(String video);
}
```

可以要求操作影片链接库的部门，必须操作 Player 完成想要的功能，而你可以先完成你的动画播放。如：

Reflection MediaMaster.java

```
package cc.openhome;

import java.util.Scanner;

public class MediaMaster {
    public static void main(String[] args) throws ReflectiveOperationException {
        String playerImpl = System.getProperty("cc.openhome.PlayerImpl");
        Player player = (Player) Class.forName(playerImpl)
                                     .getDeclaredConstructor().newInstance();
        System.out.print("输入想播放的影片：");
        player.play(new Scanner(System.in).nextLine());
    }
}
```


严禁网络传播本PDF，违者责任自负！

在这个程序中，没有写死操作 Player 的类名称，这可以在启动程序时，通过系统属性 cc.openhome.PlayerImpl 指定。例如，若操作 Player 的类名称为 cc.openhome. ConsolePlayer，其操作如下：

Reflection ConsolePlayer.java

```
package cc.openhome;

public class ConsolePlayer implements Player {
    @Override
    public void play(String video) {
        System.out.println("正在播放 " + video);
    }
}
```

执行 MediaPlayer 指定-Dcc.openhome.PlayerImpl=cc.openhome.ConsolePlayer，则执行结果如下：

```
输入想播放的影片：Hello! Duke!
正在播放 Hello! Duke!
```

若类定义有多个构造函数，也可以指定使用哪个构造函数生成对象，这必须在调用 Class 的 **getConstructor()**或 **getDeclaredConstructor()**方法时指定参数类型，取得代表构造函数的 Constructor 对象，再利用 Constructor 的 **newInstance()**指定创建时的参数值来建立对象。例如，假设因为某个原因，必须动态加载 java.util.List 操作类，只知道操作类会有个接受 int 的构造函数，可以指定 List 初始容量(capacity)，则可以这样创建实例：

```
Class clz = Class.forName(args[0]);                          // 动态加载.class
Constructor constructor = clz.getConstructor(Integer.TYPE);  // 取得构造函数
List list = (List) constructor.newInstance(100);             // 利用构造函数建立实例
```

提示»» 反射 API 有许多方法都接受不定长度自变量，善用的话可以让程序代码简洁不少。

如果要生成数组，该怎么做？数组的 Class 实例是由 JVM 生成的，也可以通过.class 或 getClass()取得 Class 实例。不过用户并不知道数组的构造函数为何，所以若要动态生成数组，必须使用 **java.lang.reflect.Array** 的 **newInstance()**方法。例如，以下动态生成长度为 10 的 java.util.ArrayList 数组：

```
Class clz = java.util.ArrayList.class;
Object obj = Array.newInstance(clz, 10);
```

取得数组对象之后，可以使用 **Array.set()**方法指定索引设值，或是使用 **Array.get()**方法指定索引取值，或者是比较偷懒的方式，直接当作 Object[](或已知的数组类型)使用：

```
Class clz = java.util.ArrayList.class;
Object[] objs = (Object[]) Array.newInstance(clz, 10);
objs[0] = new ArrayList();
ArrayList list = (ArrayList) objs[0];
```

为何要使用 Array.newInstance()建立数组实例？因为以上程序片段，obja 参考的数组实例，每个索引处都是 ArrayList 类型，而不是 Object 类型，这有什么差别？

严禁网络传播本PDF，违者责任自负！

回顾一下 9.1.7 节中操作过的 ArrayList，如果现在为其设计一个 toArray() 方法：

```
public class ArrayList<E> {
    private Object[] elems;
    ...
    public ArrayList(int capacity) {
        elems = new Object[capacity];
    }
    ...
     public E[] toArray() {
         return (E[]) elems;
     }
}
```

看来很完美不是吗？如果现在有个用户这么使用 ArrayList，悲剧就发生了：

```
ArrayList<String> list = new ArrayList<>();
list.add("One");
list.add("Two");
String[] strs = list.toArray();
```

这个程序片段会抛出 java.lang.ClassCastException，告诉你不可以将 Object[] 当作 String[] 来使用。为何？回顾一下程序片段中粗体字部分，你建立的对象确实是 Object[]，而不是 String[]，可以这样解决：

Reflection ArrayList.java

```
package cc.openhome;

import java.lang.reflect.Array;
import java.util.Arrays;

public class ArrayList<E> {
    private Object[] elems;
    private int next;

    public ArrayList(int capacity) {
        elems = new Object[capacity];
    }

    public ArrayList() {
        this(16);
    }
    ...

     public E[] toArray() {
         E[] elements = null;
         if(size() > 0) {
             elements = (E[]) Array.newInstance(elems[0].getClass(), size());
             for(int i = 0; i < elements.length; i++) {
                 elements[i] = (E) elems[i];
             }
         }
         return elements;
     }
}
```


严禁网络传播本PDF，违者责任自负！

在调用 toArray()时，如果 ArrayList 收集对象长度不为 0，可以从第一个索引取得被收集对象实际的 Class 实例，此时就可以用它配合 Array.newInstance()建立数组实例。例如，实际上收集 String 对象的话，则建立的数组就会是 String[]，调用 toArray()的客户端，就不会收到 java.lang.ClassCastException 了。

17.1.5 操作对象方法与成员

17.1.3 节介绍过，java.lang.reflect.Method 实例是方法的代表对象，可以使用 **invoke()** 方法来动态调用指定的方法。例如，有个 Student 类：

Reflection Student.java

```
package cc.openhome;
public class Student {
    private String name;
    private Integer score;
    public Student() {}
    public Student(String name, Integer score) {
        this.name = name;
        this.score = score;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setScore(Integer score) {
        this.score = score;
    }
    public Integer getScore() {
        return score;
    }
}
```

以下程序片段可以动态生成 Student 实例，并通过 setName()设定名称，用 getName()取得名称：

```
Class clz = Class.forName("cc.openhome.Student");
Constructor constructor = clz.getConstructor(String.class, Integer.class);
Object obj = constructor.newInstance("caterpillar", 90);
// 指定方法名称与参数类型，调用 getMethod()取得对应的公开 Method 实例
Method setter = clz.getMethod("setName", String.class);
// 指定参数值调用对象 obj 的方法
setter.invoke(obj, "caterpillar");
Method getter = clz.getMethod("getName");
out.println(getter.invoke(obj));
```

这只是示范动态调用方法的基本流程。来看个实际应用。下面会设计一个 BeanUtil 类，可以指定 Map 对象与类名称调用 getBean()方法，这个方法会抽取 Map 内容并封装为指定类的实例，

严禁网络传播本PDF，违者责任自负！

Map 的键为要调用的 setXXX() 方法名称(不包括 set 开头的名称，例如要调用 setName() 方法，只要给定键为 name 即可)，而值为调用 setXXX() 时的自变量。

提示 »» Java 中称 setXXX() 这类的方法为设值方法(Setter)，而 getXXX() 这类的方法为取值方法(Getter)。

例如，若 Map 中收集了学生数据，则以下返回的就是 Student 实例，其中包括了 Map 的信息：

```
Map<String, Object> data = new HashMap<>();
data.put("name", "Justin");
data.put("score", 90);
Student student = (Student) BeanUtil.getBean(data, "cc.openhome.Student");
// 下面显示(Justin, 90)
out.printf("(%s, %d)%n", student.getName(), student.getScore());
```

下面为 BeanUtil 成果，相关说明直接以批注方式撰写。

Reflection BeanUtil.java

```
package cc.openhome;

import java.lang.reflect.*;
import java.util.*;

public class BeanUtil {
    public static <T> T getBean(Map<String, Object> data, String clzName)
                                    throws Exception {
        Class clz = Class.forName(clzName);
        Object bean = clz.getDeclaredConstructor().newInstance();

        data.entrySet().forEach(entry -> {
            String setter = String.format("set%s%s",
                    entry.getKey().substring(0, 1).toUpperCase(),
                    entry.getKey().substring(1));
            try {
                // 根据方法名称与参数类型取得 Method 实例
                Method method = clz.getMethod(
                        setter, entry.getValue().getClass());
                // 必须是公开方法
                if(Modifier.isPublic(method.getModifiers())) {
                    // 指定实例与参数值调用方法
                    method.invoke(bean, entry.getValue());
                }
            } catch(IllegalAccessException | IllegalArgumentException |
                    NoSuchMethodException | SecurityException |
                    InvocationTargetException ex) {
                throw new RuntimeException(ex);
            }
        });

        return (T) bean;
    }
}
```

在某些情况下，也许想调用受保护的(protected)或私有(private)方法，可以使用 Class 的

严禁网络传播本PDF，违者责任自负！

getDeclaredMethod()取得方法，并在调用 Method 的 **setAccessible()**时指定为 true，将 Java 语言的存取限制解除。例如：

```
Method priMth = clz.getDeclaredMethod("priMth", ...);
priMth.setAccessible(true);
priMth.invoke(target, args);
```

也可以使用反射机制存取类数据成员(Field)，Class 的 **getField()**可取得公开的 Field，如果想取得私有的 Field，可以使用 **getDeclaredField()**方法。例如，动态建立 Student 实例并存取 private 的 name 与 score 成员：

```
Class clz = Student.class;
Object o = clz.getDeclaredConstructor().newInstance();
Field name = clz.getDeclaredField("name");
Field score = clz.getDeclaredField("score");
name.setAccessible(true);    // 如果是 private 的 Field，要修改得调用此方法
score.setAccessible(true);
name.set(o, "Justin");
score.set(o, 90);
Student student = (Student) o;
// 下面显示(Justin 90)
out.printf("(%s, %d)%n", student.getName(), student.getScore());
```

在调用 **Method** 或 **Field** 的 **setAccessible(true)**时，如果模块权限设定上不允许，会抛出 **java.lang.reflect.InaccessibleObjectException**。就目前来说，上面的程序片段中 setAccessible(true)可以用于同一模块的类，然而不能用于 java.base 中的类，因为 java.base 模块并没有开放权限，进一步的说明，可以参考 17.1.7 节。

17.1.6 动态代理

在反射 API 中有个 Proxy 类，可动态建立接口的操作对象。在了解这个方法如何使用之前，先来看个简单的例子。如果需要在执行某些方法时进行日志记录，你可能会这样撰写：

```
public class HelloSpeaker {
    public void hello(String name) {
        // 方法执行开始时留下日志
        Logger.getLogger(HelloSpeaker.class.getName())
              .log(Level.INFO, "hello() 方法开始...");
        // 程序主要功能
        out.printf("哈啰, %s%n", name);
        // 方法执行完毕前留下日志
        Logger.getLogger(HelloSpeaker.class.getName())
              .log(Level.INFO, "hello() 方法结束...");
    }
}
```

如果希望 hello()方法执行前后都能留下日志，最简单的做法就是以上的程序片段，在方法执行前后进行日志，然而日志程序代码写死在 HelloSpeaker 类中。对于 hello()方法来说，日志的动作并不属于它本身的职责(显示“Hello”等文字)，如果程序中到处都有这种日志需求，以上写法必须到处撰写这些日志程序代码，维护日志程序代码的麻烦会加大。若有天不再需要

严禁网络传播本PDF，违者责任自负！

日志程序代码，就必须找出进行日志的程序代码加以删除，无法简单地将日志服务从已有的程序中移去。

可以使用代理(Proxy)机制来解决这个问题。在这里讨论两种代理方式：静态代理(Static proxy)和动态代理(Dynamic proxy)。

1. 静态代理

在静态代理实现中，代理对象与被代理对象必须实现同一接口，在代理对象中可以实现日志服务，必要时调用被代理对象，这样被代理对象中就可以仅撰写本身应尽的职责。举例来说，可定义一个 Hello 接口：

Reflection Hello.java

```
package cc.openhome;

public interface Hello {
    void hello(String name);
}
```

如果有个 HelloSpeaker 类操作了 Hello 接口：

Reflection HelloSpeaker.java

```
package cc.openhome;

public class HelloSpeaker implements Hello {
    public void hello(String name) {
        System.out.printf("哈啰, %s%n", name);
    }
}
```

在 HelloSpeaker 类中没有任何日志程序代码，日志程序代码会放至代理对象中，代理对象同样也要实现 Hello 接口。例如：

```
import java.util.logging.*;
public class HelloProxy implements Hello {
    private Hello helloObj;

    public HelloProxy(Hello helloObj) {
        this.helloObj = helloObj;
    }

    public void hello(String name) {
        log("hello()方法开始...");          // 日志服务
        helloObj.hello(name);               // 执行商业规则
        log("hello()方法结束...");          // 日志服务
    }

    private void log(String msg) {
        Logger logger = Logger.getLogger(HelloProxy.class.getName())
                              .log(Level.INFO, msg);
    }
}
```

在 HelloProxy 类的 hello()方法中，真正调用 Hello 的 hello()方法前后可以安排日志程序

代码。可以这样使用代理对象：

```
Hello proxy = new HelloProxy(new HelloSpeaker());
proxy.hello("Justin");
```

创建代理对象 HelloProxy 时必须指定被代理对象 HelloSpeaker，代理对象代理 HelloSpeaker 执行 hello() 方法，在实际调用 HelloSpeaker 的 hello() 方法前后加上日志，HelloSpeaker 在撰写时就不必介入日志，可以专心于本身职责。

显然地，静态代理必须为个别接口操作出个别代理类，在应用程序行为复杂时，多个接口就必须定义多个代理对象，操作与维护代理对象会有不少的负担。

2. 动态代理

反射 API 中提供动态代理相关类，可让你不必为特定接口操作特定代理对象。使用动态代理机制，可使用一个处理者(Handler)代理多个接口的操作对象。

处理者类必须操作 **java.lang.reflect.InvocationHandler** 接口，下面以实例进行说明。例如，设计一个 LoggingProxy 类：

Reflection LoggingProxy.java

```
package cc.openhome;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingProxy {

    public static Object bind(Object target) {
        return Proxy.newProxyInstance(  ←— ❶ 动态建立代理对象
                        target.getClass().getClassLoader(),
                        target.getClass().getInterfaces(),
             new LoggingHandler(target)
         );
    }

    private static class LoggingHandler implements InvocationHandler {
        private Object target;
        LoggingHandler(Object target) {
this.target = target;
}

    public Object invoke(Object proxy, Method method, ←— ❷ 代理对象的方法被调用时
                         Object[] args) throws Throwable {    会调用此方法
        Object result = null;
        try {
            log(String.format("%s() 呼叫开始...", method.getName()));←— ❸ 实现日志
            result = method.invoke(target, args); ←— ❹ 实现被代理对象职责
            log(String.format("%s() 呼叫结束...", method.getName())); ←— ❺ 实现日志
```

```
            } catch (IllegalAccessException | IllegalArgumentException |
                    InvocationTargetException e){
                log(e.toString());
            }
            return result;
        }

        private void log(String message) {
            Logger.getLogger(LoggingHandler.class.getName())
                    .log(Level.INFO, message);
        }
    }
}
```

主要概念使用 **Proxy.newProxyInstance()** 方法建立代理对象，调用时必须指定类加载器，告知要代理的接口，以及接口上定义方法被调用时的处理者(InvocationHandler 实例)❶。Proxy.newProxyInstance() 方法底层会使用原生(Native)方式生成代理对象的 Class 实例，并利用它来生成代理对象，代理对象会操作指定要代理的接口。

如果操作 Proxy.newProxyInstance() 返回的代理对象，在每次操作时会调用处理器(InvocationHandler 实例)的 **invoke()** 方法，并传入代理对象、被调用方法的 Method 实例与参数值❷。可以在 invoke() 方法中实现日志❸❺，利用被代理对象、被调用的方法 Method 实例与参数值实现被代理对象的职责❹。

提示 »» 因为 JDK9 的日志 API 被放在 java.logging 模块中，所以记得在模块描述文件加上 requires java.logging。

接下来可使用 LoggingHandler 的 bind() 方法来绑定被代理对象：

Reflection ProxyDemo.java

```
package cc.openhome;

public class ProxyDemo {
    public static void main(String[] args) {
        Hello helloProxy = (Hello) LoggingProxy.bind(new HelloSpeaker());
        helloProxy.hello("Justin");
    }
}
```

执行结果如下：

```
三月 13, 2018 16:16:58 下午 cc.openhome. LoggingProxy$LoggingHandler log
信息: hello() 调用开始...
哈啰, Justin
三月 13, 2018 16:16:58 下午 cc.openhome.LoggingProxy$LoggingHandler log
信息: hello() 调用结束...
```

提示 »» 更多有关反射 API 的介绍，可以参考以下网址(*Trail: The Reflection API*)：http://docs.oracle.com/javase/tutorial/reflect/

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

17.1.7 当反射遇上模块

对于 JDK8 以及以前版本的 JDK，就算开发者不愿意，只要链接库的使用者运用反射，就可以访问类定义的成员，这包括了公开、受保护与私有成员。事实上，在 Java 的生态圈中，有不少链接库或框架，都依赖在这种深层反射(Deep reflection)机制上运行。

JDK9 的模块是为了增强封装，若反射机制依旧可以这般自由，那么模块就失去了强封装的意义。然而，也必须考虑现存链接库或框架依赖于深层反射的现况，否则这些链接库或框架将不会考虑迁移至 JDK9。

就结论而言，模块的设计者可以决定是否允许反射。首先，如前面介绍中谈到的，当在同一个模块时，反射允许访问类定义的成员，包括了公开、受保护与私有成员，接下来为了示范跨模块的反射，请将下载的范例文件中 labs\CH17\ReflectModule 复制至 C:\workspace 下，并使用 NetBeans 开启项目，以便从练习中逐步了解跨模块下对于反射的权限控制。

在 ReflectModule 项目中有 `cc.openhome` 与 `cc.openhome.reflect` 两个模块，在 NetBeans 项目的管理下，这两个模块都会在模块路径之中，在 `cc.openhome.reflect` 模块有 `cc.openhome.reflect.Some` 类，其原始码如下所示。

ReflectModule Some.java

```
package cc.openhome.reflect;

public class Some {
    private String some;
    public Some(String some) {
        this.some = some;
    }
    public void doSome() {
        System.out.println(some);
    }
}
```

如果想要在 `cc.openhome` 模块的 `cc.openhome.Main` 类中，使用 `Class.forName()` 来取得 `Some` 的 `Class` 实例，程序代码可以如下撰写。

ReflectModule Main.java

```
package cc.openhome;

public class Main {
    public static void main(String[] args) throws ClassNotFoundException {
        Class clz = Class.forName("cc.openhome.reflect.Some");
    }
}
```

现在运行 `Main` 的话，会得到以下会错误信息：

```
Exception in thread "main" java.lang.ClassNotFoundException: cc.openhome.reflect.Some
        at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:58
2)
        at
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java
:185)
        at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:496)
        at java.base/java.lang.Class.forName0(Native Method)
        at java.base/java.lang.Class.forName(Class.java:292)
        at cc.openhome/cc.openhome.Main.main(Main.java:5)
```

1. 模块图

为什么出错呢？cc.openhome.reflect 不是在模块路径之中吗？因为一个模块在模块路径之中，只是表示 JVM 可以找到模块描述文件(module-info.class)，不代表模块图(Module graph)中有这个模块，模块图中不存在的模块，自然就找不到模块中的类。目前 cc.openhome 模块的模块描述文件没有撰写任何设定，因此目前的模块图如图 17.1 所示。

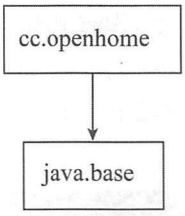


图 17.1　模块图

提示 »» 使用 NetBeans 的话，在开启 module-info.java 之后，编辑器左上方会有个 Graph 按钮，如图 17.2 所示，单击就可以看到模块图。

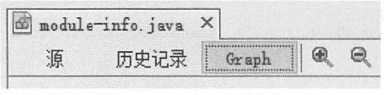


图 17.2　Graph 按钮

这个模块图表示，目前包含程序进入点的启动模块 cc.openhome 是唯一的根模块(Root module)，而它依赖在 java.base 模块。

要将模块加入模块图的方式之一，是在执行 **java** 指令启动 JVM 时，使用**--add-modules** 自变量加入其他模块作为根模块。如果执行 java 指令指定了--add-modules cc.openhome.reflect，重新执行范例，就不会出现 ClassNotFoundException，这时的模块图如图 17.3 所示。

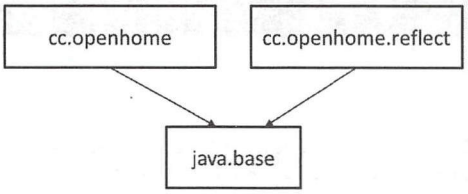


图 17.3　加入根模块

提示 »» 若使用 NetBeans，在项目上右击，在弹出的快捷菜单中选择“属性”命令，打开 Project Properties 对话框，在“类别”列表中选择 Run 选项，可以设定 Main Class 与 VM Options，如图 17.4 所示。

通过--add-modules 结合反射机制，可以让 cc.openhome 模块不依赖在 cc.openhome.reflect 的情况下加载类，--add-modules 是执行时期调整模块的弹性机制，在 19.1.2 节介绍选择性(Optional)依赖模块时，会再看到它的应用。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

另一个将指定模块加入模块图方式是，在模块描述文件中加入 requires 设定，现有模块会依赖在 requires 的模块之上，现有模块可以读取(read)该模块，或称现在模块对该模块有读取能力(Readability)。例如，在 cc.openhome 模块的 module-info.java 中设定如下：

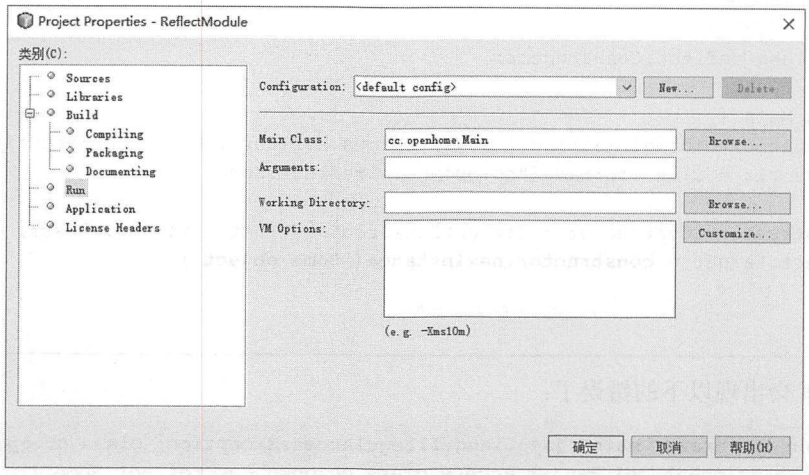


图 17.4 Project Properties 对话框

ReflectModule module-info.java

```
module cc.openhome {
    requires cc.openhome.reflect;
}
```

设定之后，重新执行范例(执行 java 指令不指定--add-modules cc.openhome.reflect)，就不会出现 ClassNotFoundException，这时的模块如图 17.5 所示。

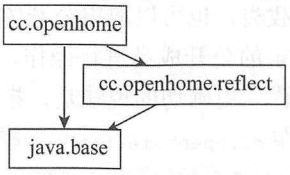


图 17.5 requires 模块

现在，进一步将 cc.openhome.Main 进行如下修改：

ReflectModule2 Main.java

```
package cc.openhome;

import java.lang.reflect.Constructor;

public class Main {
    public static void main(String[] args) throws ReflectiveOperationException {
        Class clz = Class.forName("cc.openhome.reflect.Some");

        Constructor constructor = clz.getDeclaredConstructor(String.class);
    }
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

执行程序时并不会发生问题，接着进一步使用 Constructor 的 newInstance() 来建立实例：

ReflectModule2 Main.java

```
package cc.openhome;

import java.lang.reflect.Constructor;

public class Main {
    public static void main(String[] args) throws ReflectiveOperationException {
        Class clz = Class.forName("cc.openhome.reflect.Some");

        Constructor constructor = clz.getDeclaredConstructor(String.class);
        Object target = constructor.newInstance("Some object");
    }
}
```

执行时就会出现以下的错误了：

```
Exception in thread "main" java.lang.IllegalAccessException: class cc.openhome.Main
(in module cc.openhome) cannot access class cc.openhome.reflect.Some (in module cc.
openhome.reflect) because module cc.openhome.reflect does not export cc.openhome.
reflect to module cc.openhome
    at java.base/jdk.internal.reflect.Reflection.newIllegalAccessException(Reflection.
java:361)
    at java.base/java.lang.reflect.AccessibleObject.checkAccess(AccessibleObject.
java:589)
    at java.base/java.lang.reflect.Constructor.newInstance(Constructor.java:479)
    at cc.openhome/cc.openhome.Main.main(Main.java:6)
```

2. exports 包

为什么出错了呢？虽然可以加载类，也可以取得公开的构造函数的 Constructor 实例，但是，如果模块能允许其他模块对指定的公开成员进行操作，必须在模块描述文件使用 **exports** 定义哪些包中的类是可以允许此动作。更确切的说法是，指定此模块中哪些包中的公开成员可以存取(Accessibility)，因此你必须在 cc.openhome.reflect 模块的 module-info.java 中如下撰写：

ReflectModule2 module-info.java

```
module cc.openhome.reflect {
    exports cc.openhome.reflect;
}
```

就可以允许其他模块在反射时可操作指定的公开成员，例如现在将 cc.openhome.Main 修改如下：

ReflectModule2 Main.java

```
package cc.openhome;

import java.lang.reflect.Constructor;

public class Main {
    public static void main(String[] args) throws ReflectiveOperationException {
        Class clz = Class.forName("cc.openhome.reflect.Some");
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        Constructor constructor = clz.getDeclaredConstructor(String.class);
        Object target = constructor.newInstance("Some object");
        clz.getDeclaredMethod("doSome").invoke(target);
    }
}
```

执行时就可以顺利出现如下结果：

```
Some object
```

接着，尝试来取得私有值域的 `Field` 实例：

ReflectModule3 Main.java

```
package cc.openhome;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;

public class Main {
    public static void main(String[] args) throws ReflectiveOperationException {
        Class clz = Class.forName("cc.openhome.reflect.Some");

        Constructor constructor = clz.getDeclaredConstructor(String.class);
        Object target = constructor.newInstance("Some object");
        clz.getDeclaredMethod("doSome").invoke(target);

        Field field = clz.getDeclaredField("some");
    }
}
```

执行时并没有问题，若是要取得私有值域的值会怎样呢？

ReflectModule3 Main.java

```
package cc.openhome;

import static java.lang.System.out;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;

public class Main {
    public static void main(String[] args) throws ReflectiveOperationException {
        Class clz = Class.forName("cc.openhome.reflect.Some");
        Constructor constructor = clz.getDeclaredConstructor(String.class);
        Object target = constructor.newInstance("Some object");
        clz.getDeclaredMethod("doSome").invoke(target);

        Field field = clz.getDeclaredField("some");
        field.setAccessible(true);
        out.println(field.get(target));
    }
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

执行时就会出现以下的错误信息：

```
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable to
make field private java.lang.String cc.openhome.reflect.Some.some accessible: module
cc.openhome.reflect does not "opens cc.openhome.reflect" to module cc.openhome
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(AccessibleOb
        ject.java:337)
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(AccessibleOb
        ject.java:281)
        at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:176)
        at java.base/java.lang.reflect.Field.setAccessible(Field.java:170)
        at cc.openhome/cc.openhome.Main.main(Main.java:15)
```

3. opens 套件或 open 模块

若模块能允许其他模块在反射时，对指定的非公开成员进行操作，必须在模块描述文件使用 **opens** 定义哪些包中的类是可以允许此动作，因此必须在 `cc.openhome.reflect` 模块的 module-info.java 中如下撰写：

ReflectModule3 module-info.java

```
module cc.openhome.reflect {
    opens cc.openhome.reflect;
}
```

再次执行程序的话，就可以出现以下的结果了：

```
Some object
Some object
```

除了使用 **opens** 指定要开放的包之外，也可以使用 **open module**，表示开放这个模块中全部的包，如果使用了 **open module**，那么 **module** 中的定义就不能再有 **opens** 的独立设定(因为已经开放整个模块了)。被设定为 **open module** 的模块被称为开放模块(Open Module)，相对地，没有 **open** 的模块称为一般模块(Normal module)，它们都属于显式模块。

提示 »» 命名模块、未命名模块、显式模块、自动模块、一般模块、开放模块，你到现在看过几种模块类型了？是否理清它们之间的差异性了？别担心！在 19.1.1 节会做个总结！

因此，在 `cc.openhome.reflect` 模块的 module-info.java 中如下撰写，程序也可以顺利执行：

ReflectModule3 module-info.java

```
open module cc.openhome.reflect {
    exports cc.openhome.reflect;
}
```

java.base 模块中的包使用了 **exports**，然而没有设定 **opens** 的包，也没有直接 **open module java.base**，因此若采取模块化设计，就不允许其他模块在反射时，对指定的非公开成员进行操作。

提示 »» 在采取模块化设计之后，真的得对 `java.base` 模块做深层反射，若使用 Oracle JDK，其实还是有办法的，详情可查看 19.1.4 节。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

17.1.8 使用 ServiceLoader

在 17.1.4 节曾经看过如何使用反射动态生成对象，让 MediaMaster 在撰写程序时，可以不用在意实际的 Player 实现类是什么。如果 Player 是种 API 服务，更实际的场景是，Player 会是服务的接口之一，它也许是定义在 cc.openhome.api 包之中，而 ConsolePlayer 是服务的具体实现，可能定义在 cc.openhome.impl 包之中，而 MediaMaster 是你正在设计的应用程序，定义在 cc.openhome 包之中。

在 JDK8 或更早版本 JDK 中，就算这些类与接口等实际来自于不同的 JAR 文件(也就是不同的类路径)，17.1.4 节的范例也是可以运行。然而若是采取模块设计，例如，Player 位于 cc.openhome.api 模块，ConsolePlayer 位于 cc.openhome.impl 模块，而 MediaMaster 位于 cc.openhome 模块，为了便于接下来的练习，读者可以将下载的范例文件中 labs\CH17\ ServiceLoaderDemo 复制至 C:\workspace 下，这个项目已经将 17.1.4 的范例拆成三个模块了。

为了能执行出 17.1.4 节的范例结果，你会怎么做呢？按照 17.1.7 节的介绍，在 cc.openhome.impl 模块的模块描述文件加上 exports cc.openhome.impl，然后在 cc.openhome 模块的模块描述文件加上 requires cc.openhome.impl。不过，cc.openhome 模块就依赖在 cc.openhome.impl 模块了，如果希望将来服务的实现模块是可以替换，而且不用修改 cc.openhome 模块的依赖关系的话，这就不是个好主意。

可以使用 **java.util.ServiceLoader** 来解决这个问题。首先，在 cc.openhome.api 模块中新增一个 PlayerProvider 定义。如：

ServiceLoaderDemo PlayerProvider.java

```
package cc.openhome.api;

import java.util.Optional;
import java.util.ServiceLoader;

public interface PlayerProvider {
    Player player();

    public static Player providePlayer() {
        return ServiceLoader.load(PlayerProvider.class)
                    .findFirst()
                    .orElseThrow(() -> new RuntimeException("没有服务提供商"))
                    .player();
    }
}
```

ServiceProvider 会寻找各模块中，是否有 PlayerProvider 的具体操作，并运用反射建立实例。然而基于效率与定义明确起见，必须在 cc.openhome.api 模块的 module-info.java 中，使用 **uses** 来设定这个模块会使用哪个接口提供服务。如：

ServiceLoaderDemo module-info.java

```
module cc.openhome.api {
    exports cc.openhome.api;
    uses cc.openhome.api.PlayerProvider;
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

模块描述文件中允许 import 语句，必要时也可以如下撰写：

```
import cc.openhome.api.PlayerProvider;
module cc.openhome.api {
    exports cc.openhome.api;
    uses PlayerProvider;
}
```

接着在 cc.openhome.impl 模块中，新增一个 ConsolePlayerProvider 操作 PlayerProvider，以提供具体的 Player 实例。如：

ServiceLoaderDemo ConsolePlayerProvider.java

```
package cc.openhome.impl;

import cc.openhome.api.Player;
import cc.openhome.api.PlayerProvider;

public class ConsolePlayerProvider implements PlayerProvider {
    @Override
    public Player player() {
        return new ConsolePlayer();
    }

}
```

基于效率与定义明确起见，Java 模块系统会扫描模块中具有 provides 语句的模块，看看是否有符合 uses 指定的 API 操作，因此在 cc.openhome.impl 模块的 module-info.java 中，必须使用 **privides** 设定此模块为 cc.openhome.api.PlayerProvider 提供的操作类。如：

ServiceLoaderDemo module-info.java

```
module cc.openhome.impl {
    requires cc.openhome.api;
    provides cc.openhome.api.PlayerProvider
            with cc.openhome.impl.ConsolePlayerProvider;
}
```

类似地，也可以使用 import 语句来让 provides 的部分更简洁：

```
import cc.openhome.api.PlayerProvider;
import cc.openhome.impl.ConsolePlayerProvider;
module cc.openhome.impl {
    requires cc.openhome.api;
    provides PlayerProvider with ConsolePlayerProvider;
}
```

在这样的设定之下，cc.openhome.api 模块也没有依赖在 cc.openhome.impl 模块之中，至于 cc.openhome 模块也不用 requires cc.openhome.impl 模块，只要使用以下的程序代码就可以了：

ServiceLoaderDemo module-info.java

```
package cc.openhome;

import cc.openhome.api.Player;
import cc.openhome.api.PlayerProvider;
import java.util.Scanner;

public class MediaMaster {
    public static void main(String[] args) throws ReflectiveOperationException {
        Player player = PlayerProvider.providePlayer();
        System.out.print("输入想播放的影片：");
        player.play(new Scanner(System.in).nextLine());
    }
}
```

在第16章中，实际上就使用过 `java.sql.Driver` 服务，查看 `java.sql` 模块的 module-info.java 定义，就可以看到以下的内容：

```
module java.sql {
    requires transitive java.logging;
    requires transitive java.xml;

    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;

    uses java.sql.Driver;
}
```

提示 »» `ServiceLoader` 是从 JDK6 开始就存在的 API，在基于类路径的情境下，也可以使用 `ServiceLoader`，以便为服务提供可抽换的操作，又不用接触反射的细节，方式是在服务操作的 JAR 中 META-INF/services 文件夹，放入与服务 API 类全名相同名称的文件，其中写入操作对象的类全名。

基于兼容性，服务操作的 JAR 中 META-INF/services 文件夹，若有这样的文件，而 JAR 被放在模块路径中成为自动模块，那就等同于使用了 `provides` 语句，而服务 API 的 JAR 若被放在模块路径中成为自动模块，等同于可使用任何可取得的 API 服务。

更多详情可查看 `ServiceLoader` 的 API 文件的说明：
docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html

17.2 了解类加载器

上一节介绍反射 API 时，曾不断看到类加载器这个名称。类加载器实际的职责就是载入.class 文件，JDK 本身有默认的类加载器。也可以建立自己的类加载器加入现有的加载器层级，了解类加载器层级架构，遇到 `ClassNotFoundException` 或 `NoClassDefFoundError` 就不会不知所措。

17.2.1 JDK9 类加载器层级

在 JDK9 中，有三种层次的类加载器负责加载类，分别为 System 类加载器(也称为 Application 加载器)、Platform 类加载器以及 Bootstrap 类加载器，三种类加载器有层级关系，System 的父加载器为 Platform，而 Platform 的父加载器为 Bootstrap。

注意>>> JDK8及以前版本的类加载器也是三个层级，然而与JDK9有所不同，为了避免同时包含新旧机制的内容，造成读者理解上的混乱，本书中不再赘述，可以寻找《Java JDK 8学习笔记》中有关加载器的内容，或者参考在线文件：
openhome.cc/Gossip/JavaEssence/ClassLoader.html

可以通过Class实例的getClassLoader()来取得类加载器实例，类型为ClassLoader，例如：

ClassLoaderDemo ClassLoaderHierarchy.java

```
package cc.openhome;

import static java.lang.System.out;

public class ClassLoaderHierarchy {
    public static void main(String[] args) {
        Class clz = ClassLoaderHierarchy.class;
        out.println(clz.getClassLoader());
        out.println(clz.getClassLoader().getParent());
        out.println(clz.getClassLoader().getParent().getParent());
    }
}
```

执行结果如下：

```
jdk.internal.loader.ClassLoaders$AppClassLoader@3ab39c39
jdk.internal.loader.ClassLoaders$PlatformClassLoader@71be98f5
null
```

System类加载器可以加载应用程序模块路径、类路径上的类(以及放在**jdk.javadoc**、**jdk.jartool**等模块中的JDK特定工具类)，由于范例的ClassLoaderHierarchy.class是位于模块路径中，因此会由System类加载器加载，也就是执行结果中看到的AppClassLoader。

System类加载器的父加载器为Platform加载器，因此System类加载器getParent()的显示结果是PlatformClassLoader，它可以加载整个Java SE平台API、实现类(以及特定的JDK执行时期的类)。

Platform类加载器的父加载器为Bootstrap加载器，Bootstrap类加载器是JVM内置的加载器。在以前版本JDK都是JVM源代码的实现，在Java中没有实例可作为代表，因此试图取得Bootstrap加载器实例的话会是null。在JDK9中，Bootstrap类加载器JVM源代码的实现，也有Java实现的部分，然而为了维持兼容性，试图取得Bootstrap加载器实例的结果也会是null。Bootstrap可以加载**java.base**、**java.logging**、**java.prefs**和**java.desktop**模块中的类(以及一些JDK模块)。

在JDK9中启动应用程序之后，若System类加载器需要加载类，它会先搜寻各类加载器定义的模块，如果某个类加载器定义的模块适用，就会使用该加载器来加载类。因此，若Bootstrap定义的模块适用，System类加载器可以直接委托给Bootstrap来加载类，否则委托父加载器Platform类试着加载类，如果找不到类，那么System类加载器会搜寻模块路径与类路径来试着加载类；若还是没有找到，会抛出ClassNotFoundException。

提示》》 在 JDK9 中，若执行 java 时使用--Xbootclasspath/a，在 Bootstrap 定义的模块中找不到类时，会在--Xbootclasspath/a 指定的路径中寻找，如果找到类的话，该类会被归类在未具名模块之中。

在 Platform 类加载器试着加载类时，也是先搜寻各类加载器定义的模块，如果某个类加载器定义的模块适用，就会使用该加载器来加载类。因此，若 System 定义的模块适用，Platform 类加载器也可以直接委托给 System 来加载类，否则使用父加载器 Platform 类试着加载类，如果找不到类，那么 Platform 搜寻自身定义下的模块。

例如，下面这个简单的程序，可以在 JDK9 中正常运行，其中 `cc.openhome.Some` 也定义在 `cc.openhome` 模块之中。

ClassLoaderDemo PlatformLoaderDemo.java

```
package cc.openhome;

public class PlatformLoaderDemo {
    public static void main(String[] args) throws ClassNotFoundException {
        ClassLoader platform = PlatformLoaderDemo.class
                                    .getClassLoader().getParent();
        System.out.println(platform);

        Class clz = platform.loadClass("cc.openhome.Some");
        System.out.println(clz.getClassLoader());
    }
}
```

从执行结果中可以看出，虽然取得 Platform 类加载器并试着加载指定的类，然而，Platform 类加载器委托给 System 加载器了：

```
jdk.internal.loader.ClassLoaders$PlatformClassLoader@71be98f5
jdk.internal.loader.ClassLoaders$AppClassLoader@3ab39c39
```

提示》》 在 JDK8 或旧版本 JDK 中，父加载器不能直接委托给子加载器，因此上面这个范例在旧版 JDK 中会抛出 `ClassNotFoundException`。

因此在 JDK9 中加载类时，各加载器可以直接委托的对象方向如图 17.6 所示。

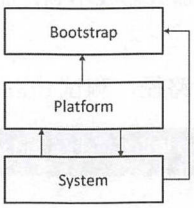


图 17.6 JDK9 类加载器阶层

如果想要直接取得 System 或 Platform 类加载器，在 JDK9 中，`ClassLoader` 增加了 `static` 的 `getSystemClassLoader()` 与 `getPlatformClassLoader()` 方法。

注意>>> 在JDK9中Platform加载器取代了旧版本JDK中的Extended加载器，也不再支持扩充机制。另外，在过去版本的JDK中，Extended与System加载器是java.net.URLClassLoader的实例。然而，在之前的范例执行结果中可以看到，JDK9中Platform与System加载器是JDK内部定义的类，如果已有的应用程序依赖URLClassLoader来操作Extended与System加载器，在JDK9中就会发生错误。

17.2.2 建立ClassLoader实例

Bootstrap加载器、Platform加载器与System加载器在程序启动后，就无法再改变它们的搜索路径。如果在程序运行过程中，打算动态决定从其他路径加载类，就要产生新的类加载器。

可以使用URLClassLoader来产生新的类加载器，它需要java.net.URL作为其参数来指定类加载的搜索路径，例如：

```
URL url = new URL("file:/c:/workspace/classes");
ClassLoader loader = new URLClassLoader(new URL[] {url});
Class clz = loader.loadClass("cc.openhome.Other");
```

使用以上方法建立URLClassLoader实例后，会设定父加载器为System Loader，URLClassLoader也有可指定父加载器的构造函数版本。使用URLClassLoader的loadClass()方法加载指定类时，会先委托父加载器代为搜索。所以上例搜索Some.class时，会一路往上委托至Bootstrap Loader，然后依Extended Loader、System Loader的路径设定顺序搜索，如果都找不到，才使用新建的URLClassLoader实例，在指定路径中搜索。看看对应的包层级下，是否有Other.class存在，若找到就载入，这时必然不是在模块路径的模块中找到类，因而被载入的类会被归类为未命名模块。虽然显式模块不能**requires**未命名模块，然而未命名模块开放全部的包，执行时显式模块可通过反射来存取。

每个**ClassLoader**实例都会有个未命名模块，可以通过**ClassLoader**实例的**getUnnamedModule()**来取得代表未命名模块的**Module**实例。

由同一类加载器载入的.class文件，只会有一个Class实例。如果同一.class文档由两个不同的类加载器载入，则会有两份不同的Class实例。

注意>>> 如果有两个自行建立的ClassLoader实例尝试搜索相同类，而在父加载器层级中就可找到指定类，就只会有一个Class实例。因为两个自行建立的ClassLoader实例都是在委托父加载器时找到类，如果父加载器找不到，而是由各自ClassLoader实例搜索到，则会有两份Class实例。

以下是个简单示范，可以指定加载路径，测试Class实例是否为同一对象。

Reflection URLClassLoaderDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class ClassLoaderDemo {
    public static void main(String[] args) {
```

```
        try {
            String path = args[0];     // 测试路径
            String clzName = args[1]; // 测试类

            Class clz1 = loadClassFrom(path, clzName);
            out.println(clz1);
            Class clz2 = loadClassFrom(path, clzName);
            out.println(clz2);

            out.printf("clz1 与 clz2 为%s 实例",clz1 == clz2 ? "相同" : "不同");
        } catch (ArrayIndexOutOfBoundsException e) {
            out.println("没有指定类加载路径与名称");
        } catch (MalformedURLException e) {
            out.println("加载路径错误");
        } catch (ClassNotFoundException e) {
            out.println("找不到指定的类");
        }
    }

    private static Class loadClassFrom(String path, String clzName)
            throws ClassNotFoundException, MalformedURLException {
        ClassLoader loader = new URLClassLoader(new URL[] {new URL(path)});
        return loader.loadClass(clzName);
    }
}
```

可以任意设计一个类，其中 path 可以输入不在 JDK 类加载器层级搜索路径的其他路径，如 file:/c:/workspace/classes/，这样同一类会分别由 loader1、loader2 参考的实例加载，结果会有两份 Class 实例，执行就会显示“clz1 与 clz2 为不相同实例”。

执行程序时，如果模块路径中的模块也包含指定的类，由于 loader1、loader2 参考的实例，父加载器会是同一个 System 加载器，System 加载器会在模块路径中的模块先找到指定类，所以最后只有一个指定类的 Class 实例，则执行就会显示“clz1 与 clz2 为相同实例”。

17.3 重点复习

Java 真正需要某个类时才会加载对应的.class 文档，而非在程序启动就加载所有类。java.lang.Class 的实例代表 Java 应用程序运行时加载的.class 文档。可以通过 Object 的 getClass()方法，或者通过.class 常量(Class literal)取得每个对象对应的 Class 对象，如果是基本类型，也可以使用对应的打包类加上.TYPE 取得 Class 对象。例如，Integer.TYPE 可取得代表 int 的 Class 对象。

使用 Integer.TYPE 取得代表 int 基本类型的 Class，如果要取得代表 Integer.class 文档的 Class，则必须使用 Integer.class。

编译时期若使用到相关类，编译程序会检查对应的.class 文档中记载的信息，以确定是否可完成编译。执行时期使用某类时，会先检查是否有对应的 Class 对象，如果没有，会加载对应的.class 文档并生成对应的 Class 实例。每个类的实例都会知道自己由哪个 Class 实例生成。默认使用 getClass()或.class 取得的 Class 实例会是同一个对象。

使用 Class.forName()方法可以实现动态加载类，Class.forName()方法在找不到指定类时会抛出 ClassNotFoundException 异常。

Class 对象代表加载的.class 文档，取得 Class 对象后，就可以取得与.class 文档中记载的信息，如包、构造函数、方法成员、数据成员等类型。每个类型会有对应的类型，例如包对应类型是 java.lang.Package，构造函数对应类型是 java.lang.reflect. Constructor，方法成员对应类型是 java.lang.reflect.Method，数据成员对应类型是 java.lang.reflect.Field 等。

如果事先不知道类名称，可以利用 Class.forName()动态加载.class 文档，取得 Class 对象之后，利用 getConstructor()取得 Constructor 实例，再调用其 newInstance()方法建立类实例。

若要动态生成数组，必须使用 java.lang.reflect.Array 的 newInstance()方法。

java.lang.reflect.Method 实例是方法的代表对象，可以使用 invoke()方法来动态调用指定的方法。

一个模块在模块路径之中，只是表示 JVM 可以找到模块描述文件(module-info.class)，不代表模块图中有这个模块，模块图中不存在的模块，自然就找不到模块中的类。

若模块能允许其他模块对包中的公开成员进行操作，必须在模块描述文件使用 exports 定义哪些包中的类允许此动作。更确切的说法是，指定此模块中哪些套中的公开成员可以存取。

若模块能允许其他模块在反射时，对指定的非公开成员进行操作，必须在模块描述文件使用 opens 定义哪些包中的类允许此动作。

除了使用 opens 指定要开放的包之外，也可以使用 open module，表示开放这个模块中全部的包。如果使用了 open module，那么 module 中的定义就不能再有 opens 的独立设定(因为你已经开放整个模块了)。被设定为 open module 的模块被称为开放模块，相对地，没有 open 的模块称为一般模块，它们都属于显式模块。

java.base 模块中的包使用了 exports，然而没有设定 opens 的包，也没有直接 open module java.base，因此不允许其他模块在反射时，对指定的非公开成员进行操作。

在 JDK9 中，有三种层次的类加载器负责加载类，分别为 System 类加载器(也称为 Application 加载器)、Platform 类加载器以及 Bootstrap 类加载器，三种类加载器有层级关系，System 的父加载器为 Platform，而 Platform 的父加载器为 Bootstrap。

在 JDK9 中启动应用程序之后，若 System 类加载器需要加载类，它会先搜寻各类加载器定义的模块，如果某个类加载器定义的模块适用，就会使用该加载器来加载类。因此，若 Bootstrap 定义的模块适用，System 类加载器可以直接委托给 Bootstrap 来加载类，否则委托父加载器 Platform 类试着加载类，如果找不到类，那么 System 类加载器会搜寻模块路径与类路径来试着加载类，若还是没有找到，会抛出 ClassNotFoundException。

在 Platform 类加载器试着加载类时，也是先搜寻各类加载器定义的模块，如果某个类加载器定义的模块适用，就会使用该加载器来加载类。因此，若 System 定义的模块适用，Platform 类加载器也可以直接委托给 System 来加载类，否则使用父加载器 Platform 类试着加载类，如果找不到类，那么 Platform 搜寻自身定义下的模块。

由同一类加载器载入的.class 文档，只会有一个 Class 实例。如果同一.class 文档由两个不同的类加载器载入，则会有两份不同的 Class 实例。

17.4 课后练习

有个对象，它是什么类的实例你一无所知，它操作了哪些接口你也不知道，你只知道对象上会有 quack() 方法，该怎么写程序来调用执行这个方法？

自定义泛型、枚举与注释

Chapter 18

学习目标

- 进阶自定义泛型
- 进阶自定义枚举
- 使用标准注释
- 自定义与读取注释

18.1 自定义泛型

在 9.1.5 节中曾经简介过基本的泛型语法，请你在继续学习之前，先复习一下该节内容。实际上泛型定义可以相当复杂，包括仅定义在方法上的泛型语法，用来限制泛型可用类型的 extends 与 super 关键字，?类型通配字符(Wildcard)的使用，以及如何结合这三者来模拟共变性与逆变性。

18.1.1 使用 extends 与 ?

在定义泛型时，可以定义类型的边界。例如：

```
class Animal {}
class Human extends Animal {}
class Toy {}
class Duck<T extends Animal> {}
public class BoundDemo {
    public static void main(String[] args) {
        Duck<Animal> ad = new Duck<Animal>();
        Duck<Human> hd = new Duck<Human>();
        Duck<Toy> hd = new Duck<Toy>();  // 编译错误
    }
}
```

在上例中，使用 extends 限制指定 T 实际类型时，必须是 Animal 的子类。可以使用 Animal 与 Human 来指定 T 实际类型，但不可以使用 Toy，因为 Toy 不是 Animal 的子类。

用快速排序法的例子来说明实际应用。

Generics Sort.java

```
package cc.openhome;

import java.util.Arrays;

public class Sort {
    public static <T extends Comparable<T>> T[] sorted(T[] array) {
        T[] arr = Arrays.copyOf(array, array.length);
        sort(arr, 0, arr.length - 1);
        return arr;
    }

    private static void sort(Object[] array, int left, int right) {
        if(left < right) {
            int q = partition(array, left, right);
            sort(array, left, q-1);
            sort(array, q+1, right);
        }
    }

    private static int partition(Object[] array, int left, int right) {
        int i = left - 1;
        for(int j = left; j < right; j++) {
            if(((Comparable) array[j]).compareTo(array[right]) <= 0) {
```


非卖品，仅供非商业用途或交流学习使用

```
                i++;
                swap(array, i, j);
            }
        }
        swap(array, i+1, right);
        return i + 1;
    }

    private static void swap(Object[] array, int i, int j) {
        Object t = array[i];
        array[i] = array[j];
        array[j] = t;
    }
}
```

提示>>> 关于快速排序法，可参考：
http://openhome.cc/Gossip/AlgorithmGossip/QuickSort1.htm
http://openhome.cc/Gossip/AlgorithmGossip/QuickSort2.htm
http://openhome.cc/Gossip/AlgorithmGossip/QuickSort3.htm

对象要能排序，基本上对象本身必须能比较大小，因此这个类实例要求 quick()方法传入的数组，当中每个元素必须是 T 类型，<T extends Comparable<T>>语法限制了 T 必须操作 java.lang.Comparable<T>接口。可以如下使用 quick()方法：

Generics SortDemo.java

```
package cc.openhome;

public class SortDemo {
    public static void main(String[] args) {
        String[] strs = {"3", "2", "5", "1"};
        for(String s : Sort.sorted(strs)) {
            System.out.println(s);
        }
    }
}
```

由于 String 操作了 Comparable 接口，因此可以用粗体字方式建立 Sort 实例，并使用其 quick()方法进行排序。

若 extends 之后指定了类或接口，想再指定其他接口，可以使用&连接。例如：

```
public class Some<T extends Iterable<T> & Comparable<T>> {
    ...
}
```

接着来看看在泛型中的类型通配字符?。如果定义了以下类：

Generics Node.java

```
package cc.openhome;

public class Node<T> {
    public T value;
    public Node<T> next;

    public Node(T value, Node<T> next) {
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
        this.value = value;
        this.next = next;
    }
}
```

如果有个 Fruit 类继承体系如下：

Generics Fruit.java

```
package cc.openhome;
class Fruit {
    int price;
    int weight;
    Fruit() {}
    Fruit(int price, int weight) {
        this.price = price;
        this.weight = weight;
    }
}
class Apple extends Fruit {
    Apple() {}
    Apple(int price, int weight) {
        super(price, weight);
    }
    @Override
    public String toString() {
        return "Apple";
    }
}

class Banana extends Fruit {
    Banana() {}
    Banana(int price, int weight) {
        super(price, weight);
    }
    @Override
    public String toString() {
        return "Banana";
    }
}
```

如果有以下程序片段，则会发生编译错误：

```
Node<Apple> apple = new Node<>(new Apple(), null);
Node<Fruit> fruit = apple;  // 编译错误，incompatible types
```

在这个片段中，apple 类型声明为 Node<Apple>，而 fruit 类型声明为 Node<Fruit>，那么 Node<Apple>是一种 Node<Fruit>吗？显然，编译程序认为不是，所以不允许通过编译。

如果 B 是 A 的子类，而 Node<B>可视为一种 Node<A>，则称 Node 具有共变性(Covariance)或有弹性的(Flexible)。从以上编译结果可看出，Java 的泛型并不具有共变性，不过可以使用类型通配字符?与 extends 来声明变量，使其达到类似共变性。例如，以下可以通过编译：

```
Node<Apple> apple = new Node<>(new Apple(), null);
Node<? extends Fruit> fruit = apple; // 类似共变性效果
```

在上面片段中使用了<? extends Fruit>语法，?代表 fruit 参考的 Node 对象，不知道 T 实

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

际声明为何种类型，加上 extends Fruit 表示虽然不知道 T 声明为何种类型，但一定会是 Fruit 或者其子类类型。由于 apple 被声明为 Node<Apple>，Apple 是一种 Fruit，所以可以通过编译。

以下是一个实际应用的例子。

Generics CovarianceDemo.java

```
package cc.openhome;

public class CovarianceDemo {
    public static void main(String[] args) {
        Node<Apple> apple1 = new Node<>(new Apple(), null);
        Node<Apple> apple2 = new Node<>(new Apple(), apple1);
        Node<Apple> apple3 = new Node<>(new Apple(), apple2);

        Node<Banana> banana1 = new Node<>(new Banana(), null);
        Node<Banana> banana2 = new Node<>(new Banana(), banana1);

        show(apple3);
        show(banana2);
    }

    public static void show(Node<? extends Fruit> n) {
        Node<? extends Fruit> node = n;
        do {
            System.out.println(node.value);
            node = node.next;
        } while(node != null);
    }
}
```

show()方法的目的是可以显示所有的水果节点，如果参数 n 仅声明为 Node<Fruit>类型，将只能接受 Node<Fruit>实例。由于 show()方法使用类型通配字符?与 extends 声明参数，使得 n 具备类似共变性的效果，因此 show()方法就可以接受 Node<Apple>实例，也可以接受 Node<Banana>实例。执行结果如下：

```
Apple
Apple
Apple
Banana
Banana
```

若声明?不搭配 extends，则默认为? extends Object。例如：

```
Node<?> node = null; // 相当于 Node<? extends Object>
```

以上的 node 可接受 Node<Object>、Node<Fruit>、Node<Apple>等对象，也就是只要角括号中的对象是一种 Object，都可以通过编译。

注意>>> Node<?>与声明为 Node<Object>不同，如果 node 声明为 Node<Object>，那就真的只能参考至 Node<Object>实例了，也就是以下会编译错误：

```
Node<Object> node = new Node<Integer>(1, null);
```

但以下会编译成功：

```
Node<?> node = new Node<Integer>(1, null);
```

一旦使用通配字符?与 **extends** 限制 T 的类型，就只能通过 T 声明的名称取得对象指定给

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

Object，或将 **T** 声明的名称指定为 **null**。除此之外，不能进行其他指定动作。例如：

```
Node<? extends Fruit> node = new Node<>(new Apple(), null);
Object o = node.value;
node.value = null;
Apple apple = node.value;          // 编译错误
node.value = new Apple();          // 编译错误
```

以上程序片段，只知道 value 参考的对象类型会是继承 Fruit，但实际上会是 Apple 还是 Banana 呢？如果实际上 node.value 是 Banana 实例，那指定给 Apple 类型的 apple 当然不对，所以编译错误。如果一开始建立 Node 时指定的 T 类型是 Banana，那将 Apple 实例指定给 node.value 就不符合原先要求，所以编译也是错误的。

因为 Java 的泛型语法只用在编译时期检查，执行时期的类型信息都是未知，也就是执行时期实际上只知道是 Object 类型(又称为类型抹除)。由于无法在执行时期获得类型信息，编译程序只能就编译时期看到的类型来做检查，因而造成以上的限制。

提示 »» Java 泛型在执行时期类型抹除，有时会让人感到困惑。例如，以下执行结果会是 true 或 false 呢？

```
List<Integer> list1 = new ArrayList<>();
List<String> list2 = new ArrayList<>();
System.out.println(list1.equals(list2));
```

许多人第一眼会认为是 false，但执行结果会显示 true，想知道为什么，可以参考一下文件《长角的东西怎么比》：

http://openhome.cc/Gossip/JavaEssence/GenericEquals.html

除了可将 T 声明的名称指定为 null 的情况，简单来说，对于支持泛型的类，例如前面范例的 Node 类定义，若使用? extends Fruit 时，Node 类定义中的 T 声明的名称，就只能作为数据的提供者，不能被重新设定；如果 Node 类使用 T 来声明某个方法的参数，那么声明该方法时就会编辑错误；若使用 T 来声明某个方法的返回类型，调用该方法则是没有问题的。

例如支持泛型的 java.util.List，如果有个 List<? extends Fruit> lt = new ArrayList<>()，那么 lt.add(fruit)就会引发编译错误，然而 lt.get(0)就没有问题。

既然 Java 泛型在执行时期类型抹除造成这类限制，不如就活用这个限制，记忆的口诀就是 Producer extends，如果想限定接收到的 List 实例，只能作为数据的提供者，那就使用? extends 这样的声明。

18.1.2 使用 super 与?

继续前一节的内容，如果 **B** 是 **A** 的子类，而 **Node<A>**视为一种 **Node<B>**，则称 **Node** 具有逆变性(Contravariance)。也就是说，如果以下代码段没有发生错误，则 Node 具有逆变性：

```
Node<Fruit> fruit = new Node<>(new Fruit(), null);
Node<Banana> node = fruit;  // 实际上会编译错误
```

Java 泛型并不支持逆变性，所以实际上第二行会发生编译错误。可以使用类型通配字符? 与 **super** 来声明，以达到类似逆变性的效果。例如：

```
Node<Fruit> fruit = new Node<>(new Fruit(), null);
Node<? super Banana> node = fruit;
Node<? super Apple> node = fruit;
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

就<? super Banana>语义来说，只知道 Node 的 T 实际上会是 Banana 或其父类型，而 Fruit 是 Banana 的父类型，因而可以通过编译。类似地，对<? super Apple>语义来说，T 实际上会是 Apple 或其父类型，而 Fruit 是 Apple 的父类型，因而可以通过编译。

这样的语法支持看似奇怪，可以从实际范例来了解如何支持逆变性。假设想设计一个篮子，可以指定篮中放置的物品，放置的物品会是同一种类(例如都是一种 Fruit)，并有一个排序方法，可指定 java.util.Comparator 针对篮中物品进行排序，那么请问以下泛型未填写部分该如何声明？

```
public class Basket<T> {
    public T[] things;
    public Basket(T... things) {
        this.things = things;
    }
    public void sort(Comparator<__________> comparator) {
        // 做一些排序
    }
}
```

声明为<? extends T>吗？如果是这样的话，篮中装 Apple 时，在调用 sort()方法时，就必须传入操作 Comparator<Apple>的对象。例如针对 price 比较时：

```
Basket<Apple> apples = new Basket<>(new Apple(20, 100), new Apple(25, 150));
apples.sort(new Comparator<Apple>() {
    @Override
    public int compare(Apple apple1, Apple apple2) {
        return apple1.price - apple2.price;
    }
});
```

因为声明为 Basket<Apple>，sort()方法就只接受 Comparator<? extends Apple>的对象。同样地，篮中装 Banana 时，在调用 sort()方法时，就必须传入操作 Comparator<Banana>的对象。例如：

```
Basket<Banana> bananas = new Basket<>(new Banana(30, 200), new Banana(50, 300));
bananas.sort(new Comparator<Banana>() {
    @Override
    public int compare(Banana banana1, Banana banana2) {
        return banana1.price - banana2.price;
    }
});
```

提示>>> 上面的两个 **Comparator** 操作，也可以使用 Lambda 语法，不过，为了强调类型以便进行说明，这边还是使用匿名类语法。

然而就以上两段程序范例来看，Apple 或 Banana 都是针对 price 比较，也就都是针对水果价格比较，都是针对 Fruit 继承而来的 price 比较，并不需要个别指定 Comparator<Apple>与 Comparator<Banana>两个比较器。如果希望可以有以下操作：

```
Comparator<Fruit> priceComparator = (fruit1, fruit2) -> fruit1.price - fruit2.price;
Basket<Apple> apples = new Basket<>(new Apple(20, 100), new Apple(25, 150));
apples.sort(priceComparator);

Basket<Banana> bananas = new Basket<>(new Banana(30, 200), new Banana(50, 300));
bananas.sort(priceComparator);
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

那么 sort()方法的 Comparator 泛型应该声明为<? super T>。例如：

Generics Basket..java

```
package cc.openhome;

import java.util.Arrays;
import java.util.Comparator;

public class Basket<T> {
    public T[] things;

    public Basket(T... things) {
        this.things = things;
    }

    public void sort(Comparator<? super T> comparator) {
        Arrays.sort(things, comparator);
    }
}
```

因此，对于 Basket<Banana>的声明，sort()方法的参数会是 Comparator<? super Banana>，可以接受 Comparator<Fruit>，而对于 Basket<Apple>的声明，sort()方法的参数会是 Comparator<? super Apple>，可以接受 Comparator<Fruit>。

范例中使用了 java.util.Arrays 的 sort()方法进行排序。查看 API 文件，可以发现 sort()方法的第二个参数正是 Comparator<? super T>，如图 18.1 所示。

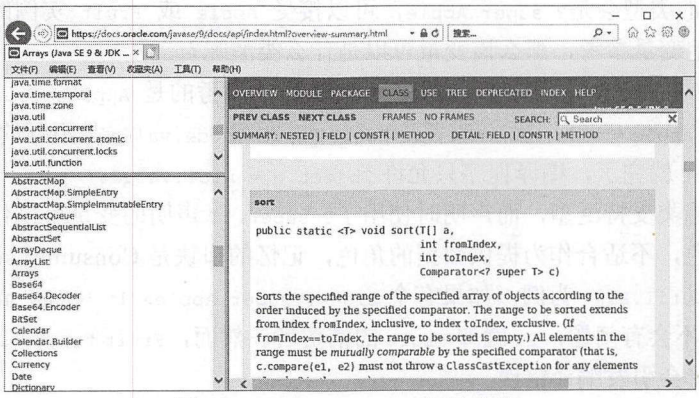


图 18.1 <? super T>实际应用

可以用以下范例看看是否能正确排序。

Generics ContravarianceDemo.java

```
package cc.openhome;

import static java.lang.System.out;
import java.util.Comparator;

public class ContravarianceDemo {
    public static void main(String[] args) {
        Comparator<Fruit> priceComparator =
                (fruit1, fruit2) -> fruit1.price - fruit2.price;
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
        Basket<Apple> apples = new Basket<>(
                new Apple(25, 150), new Apple(20, 100));
        apples.sort(priceComparator);
        for(Apple apple : apples.things) {
            out.printf("Apple(%d, %d) ",
                       apple.price, apple.weight);
        }
        out.println();

        Basket<Banana> bananas = new Basket<>(
                new Banana(50, 300), new Banana(30, 200));
        bananas.sort(priceComparator);
        for(Banana banana : bananas.things) {
            out.printf("Banana(%d, %d) ",
                       banana.price, banana.weight);
        }
        out.println();
    }
}
```

执行结果如下：

```
Apple(20, 100) Apple(25, 150)
Banana(30, 200) Banana(50, 300)
```

对于下面的程序代码：

```
Node<? super Apple> node = new Node<>(new Apple(), null);
```

node.value 的类型会是? super Apple，可以接受 Apple 或 Fruit 实例的指定。若要指定 node.value 给其他变量参考，那么该变量可以是什么类型呢？

因为类型是? super Apple，node.value 实际上可能参考的是 Apple、Fruit 或者是 Object 实例，因而，Apple apple = node.value 或者是 Fruit fruit = node.value 是不行的，如果 node.value 实际上只是 Object 就完了，编译程序只允许 Object o = node.value。

也就是说，当类支持泛型，而声明时使用了? super，T 声明的变量可以作为消费者，也就是接收指定的角色，不适合作为提供数据的角色，记忆的口诀是 Consumer super。

同样以 java.util.List 为例，如果有个 List<? super Apple> lt = new ArrayList<>()，那么 lt.add(fruit) 不会有问题，lt 就如 Fruit 的消费者。然而，Fruit fruit = lt.get(0) 或 Apple apple = lt.get(0) 会引发编译错误。

提示 »» 泛型类或接口不具共变性或逆变性，可称为不可变的(Nonvariant)或严谨的(Rigid)。

18.2 自定义枚举

在 7.2.3 节中曾经简介过枚举类型，请先了解该节内容。本节将讨论有关枚举类型的定义与运用。

18.2.1 了解 java.lang.Enum 类

在 7.2.3 节中使用 enum 定义过以下的 Action 枚举类型：

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
public enum Action {
    STOP, RIGHT, LEFT, UP, DOWN
}
```

在当时谈过，enum 定义了特殊的类，继承自 **java.lang.Enum**，不过这是由编译程序处理，直接撰写程序继承 Enum 类会被编译程序拒绝。即便如此，想要了解枚举类型如何定义与运用，先了解 Enum 类是必要的。首先看到 Enum 的 class 定义：

```
public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {
    ...
    public final int compareTo(E o) {
        Enum other = (Enum)o;
        Enum self = this;
        if (self.getClass() != other.getClass() && // optimization
            self.getDeclaringClass() != other.getDeclaringClass())
            throw new ClassCastException();
        return self.ordinal - other.ordinal;
    }
    ...
}
```

Enum 是个抽象类，无法直接实例化，它操作了 **Comparable** 接口，在 **compareTo()** 方法中，主要是针对 **ordinal** 成员比较，也就是在需要排序 Enum 实例的场合，是依据 ordinal 成员进行排序的。ordinal 成员值在 Enum 构造函数中设定：

```
    ...
    private final String name;
    private final int ordinal;

    protected Enum(String name, int ordinal) {
        this.name = name;
        this.ordinal = ordinal;
    }

    public final String name() {
        return name;
    }
    public String toString() {
        return name;
    }
    public final int ordinal() {
        return ordinal;
    }
    ...
```

还记得7.2.3节中曾列出Action.class反编译后的内容吗？以下更详细地列出其反编译后的结果：

```
public final class Action extends Enum {
    public static Action[] values() {
        return (Action[])$VALUES.clone();
    }

    public static Action valueOf(String s) {
        return (Action)Enum.valueOf(Action, s);
    }

    private Action(String s, int i) {
        super(s, i);
    }
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

```
    public static final Action STOP;
    public static final Action RIGHT;
    public static final Action LEFT;
    public static final Action UP;
    public static final Action DOWN;
    ...
    static {
        STOP = new Action("STOP", 0);
        RIGHT = new Action("RIGHT", 1);
        LEFT = new Action("LEFT", 2);
        UP = new Action("UP", 3);
        DOWN = new Action("DOWN", 4);
        ...
    }
}
```

Action 的构造函数被声明为 private，因此只能在 Action 类中调用。调用构造函数时，会传入代表 Action 枚举成员的名称字符串与 int 值，而在 Action 构造函数中调用了 super()，因此枚举成员的名称字符串与 int 值会分别设定给 Enum 的 name 与 ordinal 成员，而 **ordinal** 的值，会使用 **enum** 枚举的成员顺序，数值由 0 开始。

通过 Enum 定义的 **name()** 方法可以取得枚举成员名称字符串，这适用于需要使用字符串代表枚举值的场合，相当于 toString() 的作用。事实上，toString() 也只是返回 name 成员的值；可通过 **ordinal()** 取得枚举 int 值，这适用于需要使用 int 代表枚举值的场合。例如，在 JDK 1.4 之前撰写的 API，仍是使用 interface 定义常数作为枚举值，在使用 enum 定义枚举之后，若仍想与这些旧 API 合作，就可以调用 Enum 实例的 ordinal() 方法。例如 7.2.1 节中的 Game 类，可以如下操作：

Enum GameDemo.java

```
package cc.openhome;

public class GameDemo {
    public static void main(String[] args) {
        Game.play(Action2.DOWN.ordinal());
        Game.play(Action2.RIGHT.ordinal());
    }
}
```

如果 Action2 定义如下：

Enum Action2.java

```
package cc.openhome;

public enum Action2 {
    STOP, RIGHT, LEFT, UP, DOWN
}
```

则会有以下执行结果：

```
播放向下动画
播放向右动画
```


严禁网络传播本PDF，违者责任自负！

提示 >>> switch 比较时可以使用 Enum 类型，实际上也是利用了 Enum 的 ordinal()取得 int 值。

Enum 的 **valueOf()**方法，可以传入字符串与 Enum 实例，它会返回对应的枚举实例。例如以下会显示 true：

```
Action2 action = Enum.valueOf(Action2.class, "UP");
System.out.println(Action2.UP == action);
```

不过通常会通过 Enum 子类的 valueOf()方法，其内部就使用了 Enum.valueOf()(可观察前面反编译 Action 枚举的程序代码)。例如以下会显示 true：

```
Action2 action = Action2.valueOf("UP");
System.out.println(Action2.UP == action);
```

Enum 的 **equals()**与 **hashCode()**基本上继承了 **Object** 的行为，但被标示为 **final**：

```
...
    public final boolean equals(Object other) {
        return this==other;
    }

    public final int hashCode() {
        return super.hashCode();
    }
...
```

由于标示为 final，所以定义枚举时，不能重新操作 equals()与 hashCode()，这是因为枚举成员，在 JVM 中只会存在单一实例，Object 定义的 equals()与 hashCode()作为对象相等性比较是适当的定义。

18.2.2 enum 高级运用

观察前面反编译 Action 枚举的程序代码，可以看到还有个 **values()**方法，这个方法会将内部维护 Action 枚举实例的数组复制后返回。如果想要知道有哪些枚举成员，就可以使用这个方法。由于是复制品，因此改变返回的数组，并不会影响 Action 内部所维护的数组。

枚举类型既然继承自 Enum 的类，除了由编译程序自动产生的 private 构造函数之外，也可以自行定义构造函数，条件是不得为公开(**public**)构造函数，也不可以在构造函数中调用 **super()**。

来看个实际应用。ordinal 的值，会使用 enum 枚举的成员顺序，数值由 0 开始，如果这不是你想要的顺序呢？例如，原本有个 interface 定义的枚举常数：

```
public interface Priority {
    int MAX = 10;
    int NORM = 5;
    int MIN = 1;
}
```

现在想要使用 enum 重新定义枚举，但又必须与既存 API 搭配，也就是定义好的枚举实例，必须有个 int 值符合已存在 API 的 Priority 值，这时怎么办？可以这样定义：

Enum Priority.java

```
package cc.openhome;

public enum Priority {
    MAX(10), NORM(5), MIN(1); ←❶ 调用 enum 构造函数

    private int value;
    private Priority(int value) { ←❷ 不为 public 的构造函数
        this.value = value;
    }

    public int value() {←❸ 自定义方法
        return value;
    }

    public static void main(String[] args) {
        for(Priority priority : Priority.values()) {
            System.out.printf("Priority(%s, %d)%n", priority, priority.value());
        }
    }
}
```

在这里构造函数定义为 private❷，在 enum 中调用构造函数比较特别，直接在枚举成员后加上括号，就可以指定构造函数需要的自变量❶。由于 Enum 的 ordinal()被声明为 final，不能重新定义，所以自定义了 value()方法来返回 int 值。执行结果如下所示：

```
Priority(MAX, 10)
Priority(NORM, 5)
Priority(MIN, 1)
```

来看看 Priority.class 反编译后的结果：

```
public final class Priority extends Enum {
    ...
    private Priority(String s, int i, int value) {
        super(s, i);
        this.value = value;
    }
    public int value() {
        return value;
    }
    ...
    public static final Priority MAX;
    public static final Priority NORM;
    public static final Priority MIN;
    private int value;
    private static final Priority $VALUES[];

    static
    {
        MAX = new Priority("MAX", 0, 10);
        NORM = new Priority("NORM", 1, 5);
        MIN = new Priority("MIN", 2, 1);
        $VALUES = (new Priority[] {
            MAX, NORM, MIN
        });
    }
}
```

实际上已定义的构造函数只是编译程序用来产生真正构造函数时参考之用，而定义的 value 参数，编译程序会放在真正构造函数的 name 与 ordinal 之后，真正的构造函数会调用 super()时传入 name 与 ordinal(所以不可以在自定义构造函数中调用 super())，接着才是自定义构造函数中的程序代码。在 static 区块中，编译程序仍自行维护 name 与 ordinal 的值，接着才是调用自定义构造函数时传入的 value 值。

提示>>> 在 13.3.2 节介绍过 Month，它是 enum 类型，想要取得代表月份的数要使用 getValue()方法，而不是 ordinal()，getValue()就是自定义的方法。

定义枚举时还可以操作接口。例如，有个接口定义如下：

Enum Command.java

```
package cc.openhome;

public interface Command {
    void execute();
}
```

若要在定义枚举时操作 Command 接口，基本方式可以这样：

```
public enum Action3 implements Command {
    STOP, RIGHT, LEFT, UP, DOWN;
    public void execute() {
        switch(this) {
            case STOP:
                out.println("播放停止动画");
                break;
            case RIGHT:
                out.println("播放向右动画");
                break;
            case LEFT:
                out.println("播放向左动画");
                break;
            case UP:
                out.println("播放向上动画");
                break;
            case DOWN:
                out.println("播放向下动画");
                break;
        }
    }
}
```

基本上就是使用 enum 定义枚举时，使用 implements 操作接口，并将接口定义的方法操作，就如同定义 class 时使用 implements 操作接口。

如果在操作接口，希望各枚举成员可以有不同操作，例如上面程序片段中，其实想让枚举成员不仅有各自枚举实例，还可以带有各自的可执行指令，也就是希望可以如下这样执行程序。

Enum Game3.java

```
package cc.openhome;

public class Game3 {
    public static void play(Action3 action) {
```

```
        action.execute();
    }

    public static void main(String[] args) {
        Game3.play(Action3.RIGHT);
        Game3.play(Action3.DOWN);
    }
}
```

执行结果如下：

```
播放右转动画
播放向下动画
```

为了这个目的，前面在操作 Command 的 execute()方法时，使用 switch 比较枚举实例。但其实可以有更好的做法，就是定义 enum 时有个特定值类本体(Value-Specific Class Bodies)语法。直接来看如何运用此语法：

Enum Action3.java

```
package cc.openhome;

import static java.lang.System.out;

public enum Action3 implements Command {
    STOP {
        public void execute() {
            out.println("播放停止动画");
        }
    },
    RIGHT {
        public void execute() {
            out.println("播放右转动画");
        }
    },
    LEFT {
        public void execute() {
            out.println("播放左转动画");
        }
    },
    UP {
        public void execute() {
            out.println("播放向上动画");
        }
    },
    DOWN {
        public void execute() {
            out.println("播放向下动画");
        }
    };
}
```

可以看到在枚举成员后，直接加上{}操作 Command 的 execute()方法，这代表着每个枚举实例都会有不同的 execute()操作，在职责分配上，比 switch 的方式清楚许多。

实际上，编译程序会将 Action3 标示为抽象类：

```
public abstract class Action3 extends Enum implements Command {
    ...
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

并为每个枚举成员后的{}语法产生匿名内部类。这个匿名内部类继承了 Action3，操作了 execute()方法：

```
...
    static
    {
        STOP = new Action3("STOP", 0) {
            public void execute() {
                System.out.println("\u64AD\u653E\u505C\u6B62\u52D5\u756B");
            }
        };
        RIGHT = new Action3("STOP", 0) {
            public void execute() {
                System.out.println("\u64AD\u653E\u505C\u6B62\u52D5\u756B");
            }
        };
         ...
    }
...
```

所以每个枚举成员，实际上都参考至 Action3 的匿名子类。了解这个原理后，也就可以知道，特定值类本体语法不仅在操作接口时可以使用，也可以运用在重新定义父类方法。例如重新定义 toString()，以前面 Priority 为例，可改写为以下：

Enum Priority2.java

```
package cc.openhome;

import static java.lang.String.format;

public enum Priority2 {
    MAX(10) {
        public String toString() {
            return format("(%2d) - 最大权限", value);
        }
    },
    NORM(5) {
        public String toString() {
            return format("(%2d) - 普通权限", value);
        }
    },

 MIN(1) {
        public String toString() {
            return format("(%2d) - 最小权限", value);
        }
    };

    protected int value;
    private Priority2(int value) {
        this.value = value;
    }
    public int value() {
        return value;
    }
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
        public static void main(String[] args) {
            for(Priority2 priority : Priority2.values()) {
                System.out.println(priority);
            }
        }
}
```

执行结果如下：

```
(10) - 最大权限
( 5) - 普通权限
( 1) - 最小权限
```

18.3 关于注释

从 JDK5 之后开始支持注释(Annotation)，可以在原始码中使用注释，对编译程序提供额外编译提示，或提供应用程序执行时期可读取的组态信息。注释可以仅用于原始码，编译后留在.class 文件仅供编译程序读取或开放执行时期读取。

18.3.1 常用标准注释

Java 提供了一些标准注释，这里先介绍@Override、@Deprecated、@SuppressWarnings、@SafeVarargs 与@FunctionalInterface。

1. @Override

前面经常看到的**@Override** 就是标准注释，它在原始码中提供编译程序的信息是，被注释的方法必须是父类或接口中已定义的方法，请编译程序协助是否真的为重新定义方法。例如，在重新定义 Thread 的 run()方法时：

```
public class WorkerThread extends Thread {
    public void Run() {
        //...
    }
}
```

这个程序范例中，本来重新定义 run()方法，结果打成了 Run()方法，则变成在 WorkerThread 中定义新方法。为了避免这类错误，可以加上@Override，编译程序看到@Override 这个注释，了解必须检查父类中是否存在 Run()方法，但父类实际上并没有这个方法，所以会汇报错误，如图 18.2 所示。

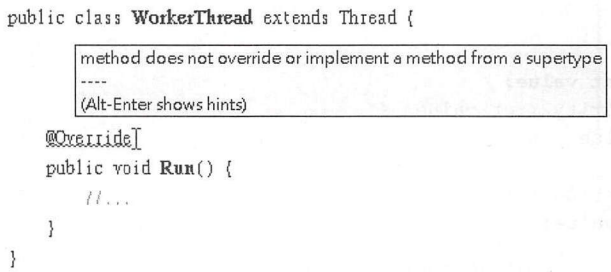


图 18.2 @Override 要求编译程序检查是否为重新定义

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

2. @Deprecated 与@SuppressWarnings

如果某个方法原先存在于 API 中，后来不建议再使用，可以在该方法上注释@Deprecated。例如：

```
public class Some {
    @Deprecated
    public void doSome() {
        ...
    }
}
```

编译后的.class 会存储这个信息，若有用户后续又想调用或重新定义这个方法，编译程序会提出警告(IDE 通常会在方法上加删除线，可参考图 11.3)。

```
Note: XXX.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

如果不想看到这个警告，使用者可以在调用该方法的场合，使用@SuppressWarnings 指定抑制 deprecation 的警告产生，例如：

```
...
    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        Some some = new Some ();
        some.doSome();
    }
...
```

在 JDK9 中增强了@Deprecated，增加了 since 与 forRemoval 两个参数，since 用来标注 API 从哪个版本弃用，默认值为空字符串，就 JDK 标准 API 的弃用来说，会使用 JDK 版本号码来标注。例如 15.4.2 节讲到 BigDecimal 上使用 int 列举的进位舍去常数，在 JDK9 之后被标示为废弃了，查看 API 文件，相关方法上就可以看到下面这类标注：

```
@Deprecated(since="9")
public BigDecimal divide(BigDecimal divisor, int roundingMode)
```

@Deprecated 的 forRemoval 默认值为 false，这样的弃用现在被归为一般弃用（Ordinary deprecation），可以使用@SuppressWarnings("deprecation")来抑制警告。若 forRemoval 被设为 true，表示这个 API 未来将会被移除，这样的弃用被称为移除弃用（Removal deprecation），@SuppressWarnings("deprecation")不会抑制这类警告，必须使用@SuppressWarnings("removal")来抑制。如果被调用的多个方法中，包含了一般弃用与移除弃用，那么想同时抑制这两种警告，必须使用@SuppressWarnings({"deprecation", "removal"})。

@SuppressWarnings 在抑制多个值时，必须使用花括号，其完整的写法应该是@SuppressWarnings(value = {"deprecation", "removal"})，@SuppressWarnings 的 value 是用来指定要抑制的警告种类。在抑制单一警告时，完整写法是@SuppressWarnings(value = {"deprecation"})，当然，简写的方式还是比较方便。

@Deprecated 可以标注在类上，表示弃用这个类，如果 import 了某个被弃用的类，那么编译程序会在 import 语句提出警告，在 JDK8 或旧版 JDK 上无法抑制这个警告，在 JDK9 中改进了这点，若是在使用到被弃用类的方法上，使用了@SuppressWarnings("deprecation")，那么就不会在 import 语句提出警告。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

提示 »» 弃用警告是由编译程序产生的，这表示用户得在新的 JDK 上重新编译原始码，才会知道已有的程序代码中是否调用了新版 JDK 中被弃用的 API。在 JDK9 中，提供了一个静态分析工具 jdeprscan，可以扫描原始码，协助找出被弃用的 API，详情可查看以下链接内容：docs.oracle.com/javase/9/tools/jdeprscan.htm

在 JDK5 之后加入泛型支持，对于支持泛型的 API，建议明确指定泛型真正类型，如果没有指定的话，编译程序会提出警告。例如程序代码含有以下片段：

```
public void doSome() {
     List list = new ArrayList();
     list.add("Some");
}
```

由于 List 与 ArrayList 支持泛型，但这里没有指定泛型真正类型，编译时会出现以下信息：

```
Note: xxx.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

如果不想看到这个警告，可以使用**@SuppressWarnings** 指定抑制 unchecked 的警告产生：

```
@SuppressWarnings("unchecked")
public void doSome() {
     List list = new ArrayList();
     list.add("Some");
}
```

3. @SafeVarargs

JDK7 增加了**@SafeVarargs** 标注，这得先谈谈一个问题，你有没有可能建立 List<String>[] 数组实例？答案是不行。无论是 new List<String>[10]或 List<String>[] lists = {new ArrayList <String>()}的语法，编译程序都会直接给你一个 error: generic array creation 的错误信息，然而声明 List<String>[] lists 是可以的，只是实际上不会有人这么做，可以声明 List<String>[] lists 是为了支持可变长度自变量，例如你可能这么声明：

```
public class Util {
    public static <T> void doSome(List<String>... varargs) {
        ...
    }
}
```

程序代码中使用了泛型声明不定长度自变量，在 JDK6 中这个程序代码可以顺利编译，也不会有任何警告。如果这么使用：

```
Util.<String>doSome("One", "Two");
```

这个程序代码可以顺利编译，也不会有任何警告。但如果这么使用：

```
List<String> list1 = new ArrayList<String>();
List<String> list2 = new ArrayList<String>();
Util.<List<String>>doSome(list1, list2);
```

编译程序会在调用 doSome()时发出警示信息：

```
>javac -Xlint:unchecked Main.java
Main.java:7: warning: [unchecked] unchecked generic array creation for varargs
parameter of type List<String>[]
        Util.doSome(list1, list2);
                    ^
1 warning
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

在 18.1.1 节介绍过，Java 泛型语法是提供编译程序信息，使其可在编译时期检查类型错误，编译程序只能就 List<String>的类型信息，在编译时期帮用户检查调用 doSome()时，传入的 list1 与 list2 是否为 List<String>类型。然而设计 doSome()的人在操作流程时，是有可能发生编译程序无法检查出来的执行时期类型错误。例如 SafeVarargs 的 API 文件中就有个范例：

```
public static <T> void doSome(List<String>... varargs) {
    Object[] array = varargs;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList;                  // 语意不对，不过编译程序不会有警告
    String s = varargs[0].get(0); // 执行时期发生 ClassCastException
}
```

这类问题称为 heap pollution，也就是编译程序无法检查执行时期的类型错误，问题是即使编译程序提醒 doSome()的客户端可能会有这类问题发生又如何？问开发 doSome()的人会不会发生问题？自己看 doSome()的源代码？最好的方式，就是对开发 doSome()的人加以提醒，而不是让使用 doSome()的人提心吊胆，因此在 JDK7 中，同样的 Util 类编译时，会发生以下警告：

```
Util.java:3: warning: [unchecked] Possible heap pollution from parameterized vararg
type List<String>
public static <T> void doSome(List<String>... varargs) {
                                              ^
1 warning
```

在使用泛型定义不定长度自变量时，编译程序会提示开发人员，有没有注意到 heap pollution 问题，这个问题是指执行时期无法具体确认(Reified)自变量类型(参数实际参考的对象类型)。如果开发人员确定避免了这个问题，则可以使用**@SafeVarargs** 加以注释。例如：

```
public class Util {
    @SafeVarargs
    public static <T> void doSome(List<String>... varargs) {
        ...
    }
}
```

加上@SafeVarargs 后，编译 Util 类就不会发生警告。由于开发人员要在确定避免了 heap pollution 的情况下，才会加上@SafeVarargs，调用 Util.doSome()的用户就不用提心吊胆，也就是这样调用 Util.doSome()就不会再发生警告。

```
List<String> list1 = new ArrayList<String>();
List<String> list2 = new ArrayList<String>();
Util.<List<String>>doSome(list1, list2);
```

提示»» 虽然在 JDK7 中，也可以这样抑制警告：

```
public class Util {
    @SuppressWarnings(value={"unchecked"})
    public static <T> void doSome(T... varargs) {
        ...
    }
}
```

不过这不仅抑制了使用泛型声明不定长度自变量的警告，连同其他 unchecked 警告也会一并抑制，所以并不建议用这个方式解决使用泛型声明不定长度自变量的警告问题。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

4. @FunctionalInterface

JDK8 为了支持 Lambda，也提出了一个**@FunctionalInterface** 标注，让编译程序可协助检查 interface 是否可做为 Lambda 的目标类型，这在 12.1.2 节已经介绍过了。

18.3.2 自定义注释类型

每个注释都会有个注释类型(Annotation Type)，所有注释类型其实都是 **java.lang.annotation.Annotation** 子接口，@Override 的注释类型为 **java.lang.Override**，@Deprecated 的注释类型为 **java.lang.Deprecated** 等。之前介绍的标准注释类型，都位于 java.lang 包中。

你可以自定义注释，先来看看如何定义标示注释(Marker Annotation)，也就是注释名称本身就是信息，对编译程序或应用程序来说，主要是检查是否有注释出现，并做出对应的动作。例如，@Override 的作用就是标示注释。要定义一个注释可以使用**@interface**。例如：

Annotation Test.java

```
package cc.openhome;
public @interface Test{}
```

编译完成后，就可以在程序代码中使用@Test 注释了。例如：

```
public class SomeTestCase {
    @Test
    public void testDoSome() {
        ...
    }
}
```

如果注释名称本身无法提供足够信息，可进一步设定单值注释(Single-value Annotation)。例如：

Annotation Test2.java

```
package cc.openhome;
public @interface Test2 {
    int timeout();
}
```

这表示注释将会有个 timeout 属性可以设定 int 值。例如：

```
@Test2(timeout = 10)
public void testDoSome2() {
    ...
}
```

注释属性也可以用数组形式指定。例如这样定义注释：

Annotation Test3.java

```
package cc.openhome;
public @interface Test3 {
    String[] args();
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

就可以用数组形式指定属性：

```
@Test3(args = {"arg1", "arg2"})
public void testDoSome3() {
    ...
}
```

在定义注释属性时，如果属性名称为 **value**，则可以省略属性名称，直接指定值。例如：

Annotation Ignore.java

```
package cc.openhome;
public @interface Ignore {
    String value();
}
```

这个注释可以使用@Ignore(value = "message")指定，也可以使用@Ignore ("message")指定，而以下这个注释：

Annotation TestClass.java

```
package cc.openhome;

public @interface TestClass {
    Class[] value();
}
```

可以使用@TestClass(value = {Some.class, Other.class})指定，也可以使用@TestClass ({Some.class,Other.class})指定。

还可以对成员设定默认值，使用 **default** 关键字即可。例如：

Annotation Test4.java

```
package cc.openhome;
public @interface Test4 {
    int timeout() default 0;
    String message() default "";
}
```

这样一来，如果设定为@Test4，则 timeout 属性默认值就是 0，message 默认就是空字符串。如果设定@Test4(timeout = 10, message = "逾时 10 秒")，则 timeout 属性的值就是 10，message 就是"逾时 10 秒"。如果是 Class 设定的属性比较特别，default 之后不能接上 null，会发生编译错误，必须自定义一个类作为默认值。例如：

Annotation Test5.java

```
package cc.openhome;
public @interface Test5 {
    Class expected() default Default.class;
    class Default {}
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

如果要设定数组默认值，可以在 default 之后加上{}。例如：

Annotation Test6.java

```
package cc.openhome;
public @interface Test6 {
    String[] args() default {};
}
```

必要时{}中可放置元素值。例如：

Annotation Test7.java

```
package cc.openhome;
public @interface Test7 {
    String[] args() default {"arg1", "arg2"};
}
```

在定义注释时，可使用 **java.lang.annotation.Target** 限定注释使用位置，限定时可指定 **java.lang.annotation.ElementType** 的枚举值：

```
package java.lang.annotation;
public enum ElementType {
    TYPE,                       // 可标注于类别、接口、列举等
    FIELD,                      // 可标注于数据成员
    METHOD,                     // 可标注于方法
    PARAMETER,                  // 可标注于方法上的参数
    CONSTRUCTOR,                // 可标注于构造函数
    LOCAL_VARIABLE,             // 可标注于局部变量
    ANNOTATION_TYPE,            // 可标注于标注类型
    PACKAGE                     // 可标注于包
    TYPE_PARAMETER,             // 可标注于类型参数，JDK8 新增
    TYPE_USE                    // 可标注于各式类型，JDK8 新增
    MODULE                      // 可标注于模块，JDK9 新增
}
```

例如想将@Test8 限定只能用于方法：

Annotation Test8.java

```
package cc.openhome;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target({ElementType.METHOD})
public @interface Test8 {}
```

尝试在方法以外的地方加上@Test8 就会发生编译错误，如图 18.3 所示。

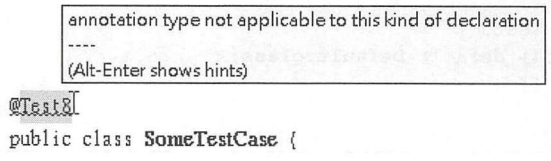


图 18.3　限定注释使用位置

在制作 JavaDoc 文件时，默认并不会将注释数据加入文件中，如果想要将注释数据加入文件，可以使用 **java.lang.annotation.Documented**。例如：

Annotation Test9.java

```
package cc.openhome;

import java.lang.annotation.Documented;

@Documented
public @interface Test9 {}
```

如果在文件中使用到@Test9，则在产生 JavaDoc 后，文件中就会包括@Test9 的信息，如图 18.4 所示。

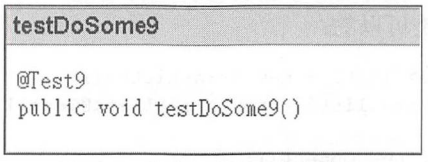


图 18.4 在文件中记录注释信息

在定义注释类型并使用于程序代码时，默认父类设定的注释，不会被继承至子类。在定义注释时设定 **java.lang.annotation.Inherited** 注释，就可以让注释被子类继承。例如：

Annotation Test10.java

```
package cc.openhome;

import java.lang.annotation.Inherited;

@Inherited
public @interface Test10 {}
```

在 JDK8 出现之前，ElementType 的枚举成员 TYPE、FIELD、METHOD、PARAMETER、CONSTRUCTOR、LOCAL_VARIABLE、ANNOTATION_TYPE、PACKAGE 等，是用来限定哪个声明位置可以进行标注。

JDK8 的 ElementType 多了两个枚举成员 **TYPE_PARAMETER**、**TYPE_USE**，它们是用来限定哪个类型可以进行标注。举例来说，如果想要对泛型的类型参数(Type parameter)进行标注：

```
public class MailBox<@Email T> {
    …
}
```

那么，在定义 @Email 时，必须在 @Target 设定 ElementType.TYPE_PARAMETER，表示这个标注可用来标注型态参数。例如：

Annotation Email.java

```
package cc.openhome;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target(ElementType.TYPE_PARAMETER)
public @interface Email {}
```

ElementType.TYPE_USE 可用于标注在各式类型，因此上面的范例也可以将 ElementType.TYPE_PARAMETER 改为 ElementType.TYPE_USE，一个标注如果被设定为 **ElementType.TYPE_USE**，只要是类型名称，都可以进行标注。例如有个标注定义如下：

Annotation Test11.java

```
package cc.openhome;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target(ElementType.TYPE_USE)
public @interface Test11 {}
```

那以下几个标注范例都是可以的：

```
List<@Test11 Comparable> list1 = new ArrayList<>();
List<? extends Comparable> list2 = new ArrayList<@Test11 Comparable>();
@Test11 String text;
text = (@Test11 String) new Object();
java.util. @Test11 Scanner console;
console = new java. util. @Test11 Scanner(System.in);
```

注意，这几个范例都仅对@Test11 右边的类型名称进行标注，必须与 JDK8 出现前就存在的枚举成员 TYPE、FIELD、METHOD、PARAMETER、CONSTRUCTOR、LOCAL_VARIABLE、ANNOTATION_TYPE、PACKAGE 等区别。举例来说，以下的标注就不合法：

```
@Test11 java.lang.String text;
```

上面这个例子中，java.lang.String text 显然是在进行 text 变量的声明，如果是在声明一个局部变量，想要让以上合法，@Test11 得在@Target 加注 ElementType.LOCAL_VARIABLE。

JDK8 除了 ElementType 多了两个枚举成员 TYPE_PARAMETER、TYPE_USE 之外，还新增了一个 **@Repeatable**，可以在同一个位置重复相同标注。举例来说，本来定义了以下的@Filter 标注：

```
public @interface Filter {
    String[] value();
}
```

这可以让你如下进行标注：

```
@Filter({"/admin", "/manager"})
public interface SecurityFilter {
    ...
}
```

如果想要另一种如下的标注风格：

Annotation SecurityFilter.java

```
package cc.openhome;

@Filter("/admin")
@Filter("/manager")
public interface SecurityFilter {}
```

在 JDK8 还没出现之前，没有办法实现这点需求，如果使用 JDK8 或之后的版本，可以如

下定义@Filter来解决这类问题：

Annotation Filter.java

```
package cc.openhome;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Filters.class)
public @interface Filter {
    String value();
}

@Retention(RetentionPolicy.RUNTIME)
@interface Filters {
    Filter[] value();
}
```

实际上这是编译程序的把戏，在这里使用@Repeatable时告诉编译程序，使用@Filters来作为收集重复标注信息的容器，而每个@Filter储存各自指定的字符串值。

在JDK9中，**ElementType**多了个枚举成员**MODULE**，而被加注**MODULE**的有**@Deprecated**和**@SuppressWarnings**，这表示模块描述文件的module上可以使用这两个标注。例如：

```
@Deprecated
module cc.openhome.simple {
    …
}
```

如果requires了被@Deprecated的模块，那么编译时会有警告，若不想看到警告，可以在模块上加注@ SuppressWarnings("deprecation")，例如：

```
@SuppressWarnings("deprecation")
module cc.openhome {
    requires cc.openhome.simple;
    …
}
```

18.3.3 执行时期读取注释信息

程序代码中若使用了自定义注释，默认会将注释信息存储于.class文档，可被编译程序或位码分析工具读取，但执行时期无法读取注释信息。如果希望在执行时期读取注释信息，可以在自定义注释时使用**java.lang.annotation.Retention**搭配**java.lang.annotation.RetentionPolicy**枚举指定：

```
package java.lang.annotation;
public enum RetentionPolicy {
    SOURCE,   // 注释信息留在原始码(不会存储至.class 文档)
    CLASS,    // 注释信息会存储至.class 文档，但执行时期无法读取
    RUNTIME   // 注释信息会存储至.class 文档，但执行时期可以读取
}
```

RetentionPolicy为SOURCE的例子为@SuppressWarnings，其作用仅在告知编译程序抑制警告信息，所以不必将这个信息存储在.class文档。@Override也是，其作用仅在告知编译程序检

查是否真为重新定义方法。

RetentionPolicy 为 RUNTIME 的时机，在于让注释在执行时期提供应用程序信息，可使用 **java.lang.reflect.AnnotatedElement** 接口操作对象取得注释信息，这个接口定义了几个方法，如图 18.5 所示。

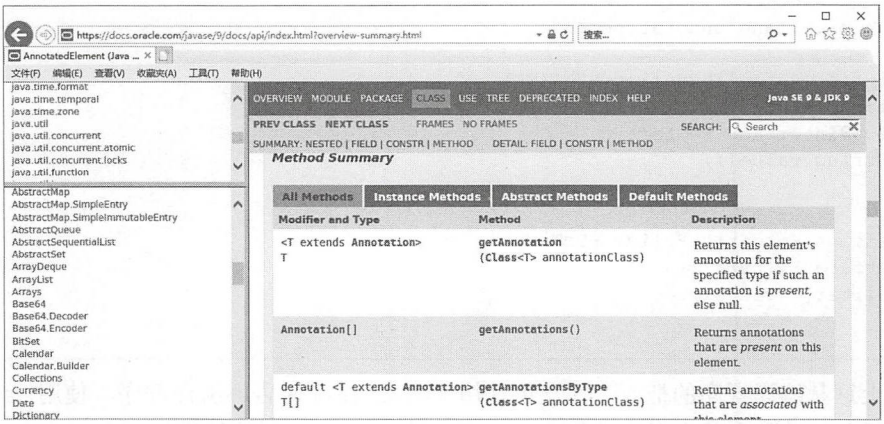


图 18.5 AnnotatedElement 接口定义的方法

Class、Constructor、Field、Method、Package 等类都操作了 AnnotatedElement 接口，如果注释在定义时的 RetentionPolicy 指定为 RUNTIME，就可以 Class、Constructor、Field、Method、Package 等类的实例，取得设定的注释信息。

举个例子来说，假设设计了以下注释：

Annotation Debug.java

```
package cc.openhome;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Debug {
    String name();
    String value();
}
```

由于 RetentionPolicy 为 RUNTIME，可以在执行时期读取注释信息。例如，可将@Debug 用于程序中：

Annotation Debug.java

```
package cc.openhome;

public class Other {
    @Debug(name = "caterpillar", value = "2018/03/10")
    public void doOther() {
        ...
    }
}
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

以下范例可用来读取@Debug 设定的信息：

Annotation DebugTool.java

```
package cc.openhome;

import static java.lang.System.out;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

public class DebugTool {
    public static void main(String[] args) throws NoSuchMethodException {
        Class<Other> c = Other.class;
        Method method = c.getMethod("doOther");
        if(method.isAnnotationPresent(Debug.class)) {
            out.println("已设定 @Debug 注释");
            showDebugAnnotation(method);
        } else {
            out.println("没有设定 @Debug 注释");
        }
        showAllAnnotations(method);
    }

    private static void showDebugAnnotation(Method method) {
        // 取得 @Debug 实例
        Debug debug = method.getAnnotation(Debug.class);
        out.printf("value: %s%n", debug.value());
        out.printf("name : %s%n", debug.name());
    }

    private static void showAllAnnotations(Method method) {
        Annotation[] annotations = method.getAnnotations();
        for(Annotation annotation : annotations) {
            out.println(annotation.annotationType().getName());
        }
    }
}
```

执行结果如下：

```
已设定 @Debug 注释
value: 2018/03/10
name : caterpillar
cc.openhome.Debug
```

从图 18.5 可以看到，JDK8 增加了 **getDeclaredAnnotation()**、**getDeclaredAnnotationsByType()**、**getAnnotationsByType()** 三个方法，getDeclaredAnnotation() 可以取回指定的标注，至于 getDeclaredAnnotationsByType() 与 getAnnotationsByType()，在指定@Repeatable 的标注时，会找寻收集重复标注的容器，相对来说，getDeclaredAnnotation() 与 getAnnotation() 就不会处理@Repeatable 的标记。使用以下范例，来读取之前看过的 SecurityFilter 上的重复的@Filter 标记信息。

Annotation SecurityTool.java

```
package cc.openhome;

import static java.lang.System.out;
```

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
    public class SecurityTool {
        public static void main(String[] args) {
            Filter[] filters = SecurityFilter.class.getAnnotationsByType(Filter.class);
            for(Filter filter : filters) {
                out.println(filter.value());
            }

            out.println(SecurityFilter.class.getAnnotation(Filter.class));
        }
}
```

执行结果如下，可以观察到，对于被标注为 @Repeatable 的 @Filter，getAnnotation()返回值会是 null：

```
/admin
/manager
null
```

18.4 重点复习

泛型也可以仅定义在方法上，可在方法返回类型前使用<T>定义泛型，之后就可以使用 T 来定义返回类型、参数类型，或在方法内声明变量、转换类型等。

在定义泛型时，使用 extends 限制指定 T 实际类型时，必须是某类的子类。

如果 B 是 A 的子类，而 Node<B>可视为一种 Node<A>，则称 Node 具有共变性(Covariance)或有弹性的(Flexible)。Java 的泛型并不具有共变性，不过可以使用类型通配字符?与 extends 来声明变量，使其达到类似共变性。

一旦使用通配字符?与 extends 限制 T 的类型，就只能通过 T 声明的名称取得对象指定给 Object，或将 T 声明的名称指定为 null。除此之外，不能进行其他指定动作。

如果 B 是 A 的子类，而 Node<A>视为一种 Node<B>，则称 Node 具有逆变性(Contravariance)。Java 泛型并不支持逆变性，可以使用类型通配字符?与 super 来声明，以达到类似逆变性的效果。

Enum 是个抽象类，无法直接实例化，它操作了 Comparable 接口，在 compareTo()方法中，主要是针对 ordinal 成员比较。ordinal 的值会是使用 enum 枚举的成员顺序，数值由 0 开始。Enum 的 equals()与 hashCode()基本上继承了 Object 的行为，但被标示为 final。

定义 enum 时可以自行定义构造函数，条件是不得为公开(public)构造函数，也不可以在构造函数中调用 super()。

定义 enum 时有个特定值类本体(Value-Specific Class Bodies)语法，可用于操作接口或重新定义父类方法。

@Override 在原始码中提供编译程序的信息是，被注释的方法必须是父类或接口中已定义的方法，请编译程序协助是否真的为重新定义方法。如果某个方法原先存在于 API 中，后来不建议再使用，可以在该方法上注释@Deprecated。可以使用@SuppressWarnings 抑制警告产生，value 属性可以指定要抑制的警告种类。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

18.5 课后练习

在 7.2.2 节中曾操作一个 ClientQueue，对 ClientQueue 中 Client 新增或移除有兴趣的对象，可以操作 ClientListener，并向 ClientQueue 注册。例如：

```
public class ClientLogger implements ClientListener {
    public void clientAdded(ClientEvent event) {
        System.out.println(event.getIp() + " added...");
    }
    public void clientRemoved(ClientEvent event) {
        System.out.println(event.getIp() + " removed...");
    }
}
```

请设计@ClientAdded 与@ClientRemoved 注释，可以注释在方法上：

```
public class ClientLogger {
    @ClientAdded
    public void clientAdded(ClientEvent event) {
        System.out.println(event.getIp() + " added...");
    }
    @ClientRemoved
    public void clientRemoved(ClientEvent event) {
        System.out.println(event.getIp() + " removed...");
    }
}
```

希望达到的功能是，如果有 Client 加入 ClientQueue，会调用@ClientAdded 注释的方法；如果有 Client 从 ClientQueue 移除，会调用@ClientRemoved 注释的方法。

提示》》 可以搭配 17.1.6 节介绍的动态代理技术。

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter 19 深入模块化

学习目标

- 认识模块的种类
- 处理模块依赖与封装细节
- 使用模块 API
- 建立 JAR、JMOD 与 JIMAGE

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

19.1 运用模块

JDK9 新增的模块化并不是语言层次的功能，有一定的复杂度，可以的话，应该是有专门负责人员来对模块进行规划、设定与管理。然而在现实环境中，身为开发者什么都要会一点，或者是必要时知道哪边可以找到相关的知识，而这一章就是为此而准备的。

19.1.1 模块的种类

先说结论，请读者不要跳过这一节！

了解模块化最好的方式，就是在必要的场景用上必要的规划与设定。因此，在前面各章节中，必要的时候已经谈过一些模块化的知识，然而分散在各章节的模块化信息有必要来个总整理，并从中一次理清模块的种类，以便后续能深入模块化。

当然，读者也可能对 Java 有相当的熟悉度，在看了这本书的目录，因为急着想了解模块化而直接翻到了这一章，建议你借助这里的总整理，看看之前在哪些章节、在什么样的场景用上了什么样的模块相关知识，这样才能进一步了解更深入的模块化问题。

在 1.2.4 节认识 JDK 安装内容时，就看到 Java SE 9 也将 JDK 本身做了模块化，src.zip 中的原始码被划分到各个模块，除了可使用 JAR 来封装模块之外，JDK9 还有 JIMAGE 与 JMOD 格式，你可以自行建立这些格式，这会在 19.2 节中介绍。

1. 命名模块、未命名模块

在 2.3 节初探模块平台系统中，介绍到了如何建立、编译、运行简单的模块，在模块描述文件的设定中，知道模块必须 `exports` 包，另一个模块在 `requires` 了该模块之后，才能使用那个包中的类；认识了模块路径，也知道了在类路径下被发现的类，都会被自动归类到未命名模块，在模块路径下被发现的类，则都属于某个命名模块。如果应用程序采取模块化设计，默认就会依赖在 **`java.base`** 模块之上，**`java.base`** 模块的包，任何模块都可以读取，如图 19.1 所示。

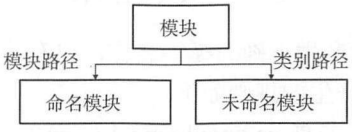


图 19.1 命名模块与未命名模块

如果会看这一章，应该表示具备包管理知识，以及 `public`、`protected`、`private` 与默认包权限。在 JDK8 或 JDK 旧版本中，`public` 表示各包间，也就是整个应用程序都可以存取，然而，在 JDK9 之后多了模块，模块中的 public 类、方法、值域是否能被另一模块看见，还要视模块设定而定。

在定义模块时必须记得，同一个包不可以同时出现在被 **`requires`** 的两个模块中，这称为分裂包(Split package)，会导致编译失败，而执行时期会产生 **`LayerInstantiationException`**。

在 2.3 节之后，各章节的范例项目都只使用模块路径，而且都只使用到 `java.base` 模块，直到 15.1 节谈到日志 API，因为 `java.util.logging` 包被划分至 `java.logging` 包模块，因此必须在模块描述文件加入 `requires java.logging`，也就是说，就算是 Java SE API，若不是 **`java.base`**

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

模块中的类，还是必须 **requires** 相关的模块；为什么 `java.util.logging` 包不放到 `java.base` 模块呢？因为通常会使用功能更强大的第三方日志链接库。

你也许只是想让既有的应用程序运行在 JDK9 上，不打算模块化，也就是从程序进入点开始的每个类，都是基于类路径，也就是都在未命名模块之中。然而，对于其他同样是放在类路径上的 JAR，在 API 的使用上也许没有问题，使用的是 `java.sql`、`java.util.logging` 包中的 API 也可以，但是若使用到 `javax.xml.bind.*`、`javax.rmi` 等包，还是会出现编译错误。

这是因为这些包虽然包含在 Java SE 中，然而实际上是与 Java EE 相关的 API，在 JDK9 中还是有这些包，不过被划分到 `java.se.ee` 模块，JVM 默认并不会加载这个模块，因此编译与执行时，必须使用`--add-modules java.se.ee`，才找得到这些包。

2. 显式模块、自动模块

许多链接库都还没有模块化，例如 16.1.2 节的 sqlite-jdbc 驱动程序，为了兼容性，是可以将这些链接库的 JAR 放在类路径之中。但是本书范例项目都是采用模块化，都明确定义了模块并放在模块路径中，这样的模块称为显式模块。显式模块无法依赖在未命名模块，因为未命名模块并没有名称，无法在显式模块的模块描述文件中进行 **requires**。

若是采用模块化设计，可以将未支持模块化设计的JAR文件放在模块路径，这会使得该JAR文件被视为自动模块，自动模块也是命名模块的一种，稍后会看到名称产生的规则。有了模块名称之后，就可以 **requires** 自动模块，也就可以使用自动模块中公开的类、方法与值域。

基于兼容性，自动模块有隐含的模块定义，可以读取其他模块，其他模块也可以读取自动模块，应用程序在迁移至模块化设计的过程中，自动模块会是未命名模块至显式模块之间的桥梁，如图 19.2 所示。

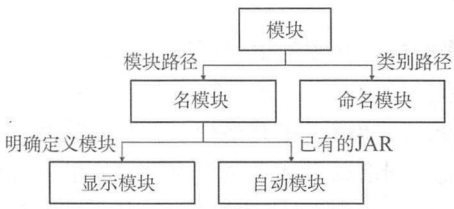


图 19.2　显示模块与自动模块

并不是任何 JAR 都可以自动产生正确的模块名称，已有的 JAR 若还没有进行任何调整，默认是基于文件名来产生名称，产生的规则如下。

- 取得 JAR 的主文件名：若是 cc.openhome-1.0.jar 的话，就取得 cc.openhome-1.0 这个名称。
- 去除版本号：版本号必须是连字符(-)或底线(_)后跟随着数字，找到版本号之后，取得连字符(-)或底线(_)前的名称，例如 cc.openhome.util-1.0 的话，就使用 cc.openhome. util；cc-openhome-util_1.0 的话，就使用 cc-openhome-util；如果没有版本号的话，例如 cc_openhome_util，就直接使用该名称。
- 替换名称中非字母部分为句号(.)：因此对于 cc.openhome.util、cc-openhome-util 或 cc_openhome_util，最后决定出来的自动模块名称都会是 cc.openhome.util。

在产生自动模块名称时，JAR 主文件名不能有多个版本号区段，例如 cc.openhome.util_1.0-spec-1.0，无法自动产生正确的模块名称，此时若被放到模块路径的 JAR 文件，编译时期没

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

有名称可以 **requires**，而执行时期会产生 **IllegalArgumentException**，从而使得 JVM 无法初始模块层而发生 **FindException**。

提示 »» 若不想基于文件名决定自动模块名称，现有的 JAR 中，可以在 META-INF/MANIFEST.MF 里增加 Automatic-Module-Name，指定自动模块名称，然而对于第三方程式库的 JAR，不建议读者自己做这个动作，最好是让第三方程式库的使用者决定自动模块名称，免得以后产生名称上的困扰。

链接库官方还没决定自动模块名称之前，依文件名来产生模块名称实际上也会引发问题，有兴趣了解的话，可参考 Java SE 9 - JPMS automatic modules 的内容，链接地址如下：

blog.joda.org/2017/05/java-se-9-jpms-automatic-modules.html

在决定自己的应用程序是否迁移至模块化之前，看看使用到的各链接库官方是不是都决定好(自动)模块名称了，可以免去后续自行修改模块名称的麻烦。

如果同一个包出现在模块路径上多个已有的 JAR 文件中，由于一个包不能出现在多个模块，因此只有其中一个 JAR 能成为自动模块，而其他会被忽略。

3. 一般模块、开放模块

第 17 章介绍了反射，由于 JDK9 支持模块化，用户在采取模块设计时，如何在不破坏模块封装下又能运用反射机制的弹性，就成了认识反射时必须知道的一大课题。就算你熟悉 Java，也对反射有一定的认识，我建议你还是要仔细阅读第 17 章，因为你过去对反射的认识，在应付同一模块中的类，以及 java.base 中的公开对象时也许足够，然而，对 java.base 模块做深层反射时就会遇上问题，而在跨模块反射时也会面临许多问题。

在 17.1.7 节中就已经探讨反射与模块的问题，而你也看到“读取能力”这个名词，若 a 模块 requires 了 b 模块，那 a 模块依赖在 b 模块，a 模块可以读取 b 模块，或说 a 模块对 b 模块是可读取的(readable)，有读取能力不代表有存取能力，b 模块还得 exports 包，a 模块对 b 模块才有存取能力，也就是可以操作公开的类、方法或值域。

比存取能力更进一步的是深层反射，如果要能操作非公开的类、方法或值域，模块本身获得 **opens** 指定的包，或者直接 **open** 模块本身。如果显示模块本身 open 了，那这个模块会是个开放模块，而没有 open 的模块会是一般模块，如图 19.3 所示。

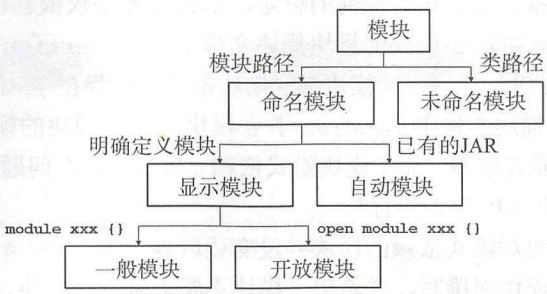


图 19.3 一般模块与开放模块

java.base 模块 **exports** 了所有的包，但没有 **opens** 包，也没有 **open** 模块本身，因此，任何模块都不能对 **java.base** 做深层反射。

前面介绍了，未命名模块可以读取其他模块，然而能不能存取或深层反射，要视其他模块是否 exports 或 opens。而未命名模块对其他模块来说，如同 open 全部包，因此执行时期，任何模块都可以存取与深层反射未命名模块。不过，因为未命名模块没有名称，因此显式模块无

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

法 requires，也就不能存取(也不能读取)未命名模块。

那么自动模块呢？自动模块可以读取其他模块，然而能不能存取或深层反射，要视其他模块是否 exports 或 opens。自动模块对其他模块来说，如同 **open** 全部包，因此可以存取与深层反射自动模块，自动模块是命名模块，显式模块也可以 **requires** 自动模块。

提示 »» 对于未命名模块，默认是找不到 javax.xml.bind.*、javax.rmi 等包，因为这些被归类为 Java EE 相关 API，位于 java.se.ee 模块之中。

17.1.8 节介绍了如何使用 **ServiceLoader**，以及如何在模块上运用 **uses**、**provides** 设定，让服务的客户端依赖在服务的 API 规范，而不用依赖在实现服务的模块上，而规范服务 API 的模块与实现服务的模块又可以保持舒服的依赖关系。

17.2 节关于类加载器的部分介绍过，因应模块化的特性，类加载器阶层有了变化，Application 也可以直接委托 Bootstrap 加载器，而 Extension 被 Platform 加载器取代，它也可以直接委托 Application 加载器；另外，如果自行建立 URLClassLoader 实例，而最后也确实是由该实例从指定路径加载了类，该类是会被归类在未命名模块之中。

应用程序依赖的模块可能被弃用，18.3.2 节中提到，在 JDK9 中，ElementType 多了列举成员 MODULE，而被加注 MODULE 的有@Deprecated 与@SuppressWarnings，这表示被弃用的模块可以使用**@Deprecated** 标注，某个模块如果还是必须使用被弃用的模块，又不想收到警告，那就加上个**@SuppressWarnings("deprecation")**。

如果对以上的内容，都已经了解的话，那么应该足以应付大多数的场合了，接下来的内容，就只是对于模块的更多细节尽量多做补充。

19.1.2 requires、exports 与 opens 细节

看到这里，你应该已经知道在模块描述文件中，requires、exports、opens 设定的基本意义是什么，它们还有一些细节可以谈谈。

1. requires transitive

如果 a 模块的模块描述文件没有任何的设定，那么 a 模块仅依赖在 java.base 模块，这是隐含(Implicit)依赖关系。如果 a 模块的模块描述文件中 requires 了 b 模块，那么 a 模块显式(Explicit)依赖在 b 模块之上。当然，a 模块与 b 模块都隐式依赖在 java.base。

如果 a 模块的模块描述文件中 requires 了 b 模块，而 b 模块的模块描述文件中 requires 了 c 模块，那么 a 对 b 显式依赖，而 a 模块隐式依赖 c 模块。现在问题来了，a 模块可以使用 c 模块 exports 的包中公开 API 吗？不行！

在默认情况下，模块对隐式依赖的模块并没有读取能力。

如果 b 模块基于 c 模块而撰写，并希望 a 模块不要去触及 c 模块，以便未来若 b 模块想替换底层实作时，不至于影响 a 模块，那这样的设计是有意义的。当然，a 模块真的想直接使用 c 模块的 API，a 模块的模块描述文件中可以设定 requires c 模块来解决这个问题，只是这时 a 模块也显式依赖 c 模块了。

然而，若 b 模块某方法返回了 c 模块中的类型，a 模块若要能使用该类型，就必须对 c 模块有读取能力。虽然可以直接在 a 模块的模块描述文件中设定 requires c 模块来解决这个问题，然而，b 模块既然揭露了 c 模块的类型，可以考虑使用 **requires transitive** c 模块，让模块在

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

requires b 模块时，对于 c 模块有隐含的读取能力，而又不用显式依赖在 c 模块。

在 Java SE API 中的例子之一，就是 java.sql 模块，你定义的模块描述文件中 requires java.sql 之后，就可以从 java.sql.Driver 的 getParentLogger()取得 java.util.logging.Logger 实例进行操作，然而后者是位于 java.logging 模块，可以这么做的原因在于，java.sql 模块描述文件是这么撰写的：

```
module java.sql {
    requires transitive java.logging;
    requires transitive java.xml;

    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;

    uses java.sql.Driver;
}
```

由于 requires transitive 了 java.logging 与 java.xml 模块，因此只要模块 requires 了 java.sql 模块，就隐式依赖在 java.logging 与 java.xml 模块，并对这两个模块隐含了读取能力。假设 cc.openhome 模块 requires 了 java.sql 模块，那么模块图如图 19.4 所示。

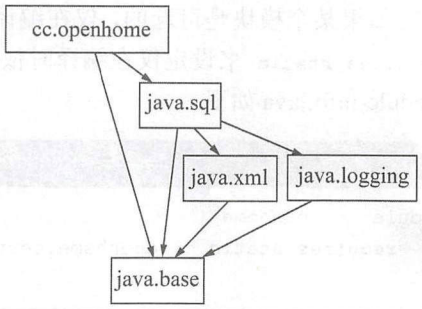


图 19.4 requires java.sql

2. requires static

requires 定义了模块间的依赖关系，而且是包含了编译时与执行时的依赖关系。在编译时，若程序代码必须使用到另一模块的公开 API，则必须 requires 该模块，编译程序才会找得到。如果 a 模块 requires 了 b 模块，在执行时 b 模块就一定得存在，否则会发生 java.lang.module.FindException，表示找不到 b 模块。

例如，可以将范例文件 labs\CH19 中的 RequiresStaticDemo 项目复制至 C:\workspace，这个项目中 cc.openhome 模块 requires 了 cc.openhome.test 模块，而 cc.openhome 模块中有个 Main 类。

RequiresStaticDemo Main.java

```
package cc.openhome;

public class Main {
    public static void main(String[] args) {
        if(args.length != 0 && "test".equals(args[0])) {
            cc.openhome.test.Test.fromTestModule();
        } else {
            System.out.println("应用程序正常流程");
        }
    }
}
```

cc.openhome.test.Test 类来自 cc.openhome.test 模块，假设它是在开发程序时测试用的类，执行 Main 时若加上命令行自变量 test 就会使用到 Test 类。在 RequiresStaticDemo 项目的 dist 中有两个 JAR，各包含了 cc.openhome 模块与 cc.openhome.test 模块，进入 dist 目录后，你可

非卖品，仅供非商业用途或交流学习使用

以这么执行它们：

```
>java -module-path cc.openhome.jar;cc.openhome.test.jar -m cc.openhome/cc.openhome.Main test
来自 cc.openhome.test 模块
```

如果没有提到命令行自变量 test，那么就会执行应用程序流程：

```
>java -module-path cc.openhome.jar;cc.openhome.test.jar -m cc.openhome/cc.openhome. Main
应用程序正常流程
```

也许 cc.openhome.test 模块在实际产品环境中并不需要，然而，由于 cc.openhome 模块 requires 了 cc.openhome.test 模块，如果没有 cc.openhome.test 模块的话就会出现错误：

```
>java --module-path cc.openhome.jar -m cc.openhome/cc.openhome.Main
Error occurred during initialization of boot layer
java.lang.module.FindException: Module cc.openhome.test not found, required by cc.openhome
```

如果某个模块是可选的，仅在编译时需要，执行时期该模块可以不用存在，那么可以使用 **requires static** 来设定仅在编译时依赖，以上面的范例来说，可以修改 cc.openhome 模块的 module-info.java 如下：

RequiresStaticDemo module-info.java

```
module cc.openhome {
    requires static cc.openhome.test;
}
```

接着在 NetBeans 中打开该项目，在项目图标上右击，在弹出的快捷菜单中选择“构建”命令，这会重新编译并构建出新的模块 JAR 文件，试着再执行一次上面指定 test 自变量的指令：

```
Exception in thread "main" java.lang.NoClassDefFoundError: cc/openhome/test/Test
        at cc.openhome/cc.openhome.Main.main(Main.java:6)
Caused by: java.lang.ClassNotFoundException: cc.openhome.test.Test
        at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClass
Loader. java:582)
        at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(Class
Loaders.java:185)
        at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:496)
        ... 1 more
```

喔？出错了？因为被 requires static 的模块仅在编译时期被依赖，执行时期并不会看到依赖关系，因而就算--module-path 上存在 cc.openhome.test.jar 文件，模块图中并不会有 cc.openhome.test 模块，这时就可以用 17.1.7 节看过的**--add-modules** 自变量了：

```
>java --add-modules cc.openhome.test --module-path cc.openhome.jar;cc.openhome.test.jar
-m cc.openhome/cc. openhome.Main test
来自 cc.openhome.test 模块
```

实际产品上线之后，因为被 requires static 的模块仅在编译时期被依赖，执行时期并不会看到依赖关系，因此可以不用 cc.openhome.test 模块：

```
>java --module-path cc.openhome.jar -m cc.openhome/cc.openhome.Main
应用程序正常流程
```


非卖品，仅供非商业用途或交流学习使用

在 **requires** 时，必要的时候 **transitive** 与 **static** 可以并用，例如：**requires transitive static** cc.openhome.somemodule;

3. exports to、opens to

模块在使用 exports 指定哪些包，还可以加上 to 限定只能被哪些模块使用，例如，java.base 模块的模块描述文件中，有些特定实现模块或 JDK 内部模块会在 exports 时限定可使用之模块：

```
module java.base {
    ...
    exports com.sun.security.ntlm to java.security.sasl;
    exports jdk.internal to jdk.jfr;
    exports jdk.internal.jimage to jdk.jlink;
    exports jdk.internal.jimage.decompressor to jdk.jlink;
    exports jdk.internal.jmod to
        jdk.compiler,
        jdk.jlink;
    ...
}
```

类似地，在使用 opens 开放包时，也可以加上 to 限定只能被哪些模块进行深层反射。

19.1.3 修补模块

有时候，你会需要临时地替换掉模块中某(些)类，例如修补 Bug，或进行测试时需要，也可能是临时性地修补分裂包的问题，则在编译或执行时，可以在 Oracle JDK 中使用非标准自变量**--patch-module** 自变量指定修补用的类来源。

提示»» 执行 javac 或 java 时，可以加上-X 查看非标准自变量。然而要注意的是，非标准属性未来可能在没有任何官方公告下移除。

例如，在范例文件的 labs\CH19 中有个 PatchModuleDemo 项目，在 dist 文件夹中项目中两个模块已创建好的 JAR 文件，如果在 dist 文件夹中执行以下指令，会显示 help 字样：

```
>java --module-path cc.openhome.jar;cc.openhome.util.jar -m cc.openhome/cc.openhome.Main
help
```

help 字样是来自于 cc.openhome.util 模块中 cc.openhome.util.Util 类的 help()方法。在 labs\CH19 的 Patch 下的 dist 文件夹中，有个 Patch.jar，其中封装了 cc.openhome.util.Util 类，并将 help()方法实现为显示 HELP 的字样，如果将 Patch.jar 复制至 PatchModuleDemo\dist 目录中，然后执行下面 java 指令：

```
>java --patch-module cc.openhome.util=Patch.jar --module-path cc.openhome.jar;cc.
openhome. util.jar-m cc.openhome/cc.openhome.Main
HELP
```

--patch-module 的指定中表示，cc.openhome.util 模块中若有符合的类，将会被 Patch.jar 中对应的类取代。

--patch-module 只能被用来替换 API 类，如果**--patch-module** 指定的来源中，含有 module-info.class，只会被忽略，并不会将之替换掉。在使用 javac 编译的时候，也可以使用 --patch-module 自变量。

非卖品，仅供非商业用途或交流学习使用

19.1.4 放宽模块封装与依赖

JDK9 的模块化功能是为了加强封装，然而在应用程序采取模块化设计之后，你会面临在不修改模块描述文件前提下，放宽已有模块封装的情况。例如打算使用未被 exports 的 JDK 内部 API，或者是过去曾对 Java SE 标准 API 做了深层反射，然后在 JDK9 模块化之后，这些 API 相对应的模块并没有 exports 或者是 opens 相关的包。

在编译时期或执行时期，若真的有必要，Oracle JDK 中可以通过非标准自变量**--add-exports**、**--add-opens** 和**--add-reads** 的指定，来增加原本各模块的模块描述文件中没有设定的 exports、opens、requires 选项。

1. --add-exports

举个例子来说，java.base 模块中的 sun.net 包或其中的子包，都只限定 exports 给 JDK 内部模块使用(jdk.*开头的模块)，如果因为某个理由，打算使用 sun.net 包或子包中的 API，例如，若在 cc.openhome 模块中有个 Main 类如下撰写：

```
package cc.openhome;

import sun.net.ftp.FtpClient;
import sun.net.ftp.impl.DefaultFtpClientProvider;

public class Main {
    public static void main(String[] args) {
        FtpClient c = new DefaultFtpClientProvider().createFtpClient();
        ...
    }
}
```

在使用 javac 编译时默认是无法通过编译的，然而若加上--add-exports 自变量，例如：

```
javac --add-exports java.base/sun.net.ftp=cc.openhome
      --add-exports java.base/sun.net.ftp.impl=cc.openhome
      ...
```

就可以通过编译，上面的--add-exports 指定，相当于在 java.base 模块的模块描述文件中增加下面的 exports to 设定：

```
exports sun.net to cc.openhome;
exports sun.net.ftp.impl to cc.openhome;
```

提示 »» 在 NetBeans 中，可以在模块项目图标上右击，在弹出的快捷菜单中选择“属性”命令，打开 Prjoect Properties 对话框，在“类别”列表中选择 Build 下的 Compiling，于 Additional Compiler Options 里可以添加编译时的额外参数如图 19.5 所示。

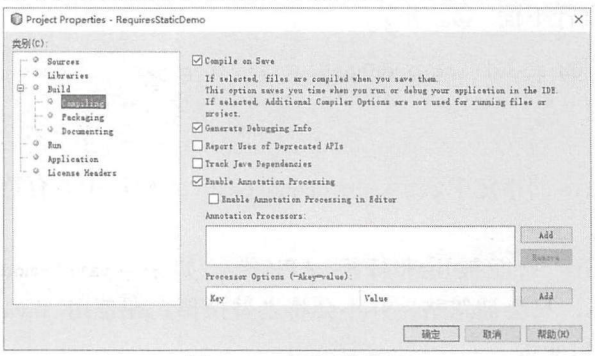


图 19.5 项目属性对话

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

如果想在--add-exports 时指定 exports 给未命名模块，等号右边可以使用 ALL-UNNAMED。在通过编译之后，执行时期使用 java 时也要指定--add-exports 自变量，才不会发生 IllegalAccessError，例如：

```
java --add-exports java.base/sun.net.ftp=cc.openhome
     --add-exports java.base/sun.net.ftp.impl=cc.openhome
     ...
```

2. --add-opens

--add-opens 的指定格式与--add-exports 相同。例如，由于 java.base 模块的 java.lang 包并没有 opens，因此下面 cc.openhome 模块中的 Main 类在执行时，会因为试图做深层反射而引发 InaccessibleObjectException：

```
package cc.openhome;
import java.lang.reflect.Field;

public class Main {
    public static void main(String[] args) throws Exception {
        String empty = "";
        Field hash = String.class.getDeclaredField("hash");
        hash.setAccessible(true);
        System.out.println(hash.get(empty));
    }
}
```

在执行 java 指令，--add-opens java.base/java.lang=cc.openhome 就可以顺利执行，因为这相当于对 java.base 模块的模块描述文件增加下面的设定：

```
opens java.lang to cc.openhome;
```

3. --add-reads

--add-reads 可以增加模块描述文件中没有定义的模块依赖关系。例如，如果 a 模块不想直接依赖在 b 模块上，因而不想在模块描述文件中直接 requires b，连 requires static b 都不想设定，但是又想使用 b 模块中 exports 的 API，这样，在使用 javac 编译的时候，就可以增加--add-reads 自变量：

```
javac --add-modules b
      --add-reads a=b
      ...
```

--add-reads a=b，相当于在 a 模块的模块描述文件中，增加 requires b 的设定。然而，实际上模块描述文件中没有 requires b 语句，为了在模块图中找得到 b 模块，记得使用--add-modules b，而使用 java 执行的时候也是如此：

```
java --add-modules b
     --add-reads a=b
     ...
```

如果将来有个 c 模块提供了 b 模块中 API 的增强版本，而执行时想要使用 c 模块来取代 b 模块，就只要将--add-modules 与--add-reads 指定中的 b 改成 c 就可以了。

提示 »» 现有的应用程序若想迁移至 JDK9，可能得费点功夫，有兴趣的话，可以参考《迁移！往 Java 9 前进！》，链接地址如下：
openhome.cc/Gossip/Programmer/MigrateToJava9.html

严禁网络传播本PDF，违者责任自负！

19.2 模块 API

JDK9 支持模块化，在执行时期，使用 java.lang.Module 实例来代表各个模块，读者可以从中取得模块的相关信息。对于模块描述文件中定义的信息，执行时期使用 java.lang.module.ModuleDescriptor 实例来代表，可以从中取得 requires、exports 等语句的代表实例，例如 ModuleDescriptor.Exports、ModuleDescriptor.Requires 等。接下来将介绍这些模块 API 以及模块层(Module layer)的概念。

19.2.1 使用 Module

在 17.1.1 节中讲过，在 JDK9 中，Class 新增了 getModule()方法，可以取得类所在的模块代表实例，类似反射 API，在取得 Module 实例之后，就可以对它进行信息的探查。如下范例可探查目前模块可否读取指定类所在之模块，并列出其 exports 的包：

ModuleAPI ModuleInfo.java

```
package cc.openhome;

import static java.lang.System.out;

public class ModuleInfo {
    public static void main(String[] args) throws Exception {
        Class clz = Class.forName(args[0]);

        Module current = ModuleInfo.class.getModule();
        Module module = clz.getModule();

        out.printf("%s 模块%s 读取 %s 类所在之 %s 模块%n",
                current.getName(),
                current.canRead(module) ? "可" : "不可",
                args[0],
                module.getName());

        out.println("exports 的包：");
        module.getPackages().stream()
                .filter(module::isExported)
                .forEachOrdered(out::println);
    }
}
```

Module 的 getName()会返回模块名称，如果是未命名模块，则会返回 null，canRead()用来判断是否可读取指定的模块。对于已命名模块，getPackages()会使用 Set<String>返回模块中包含的包，而对于未命名模块，getPackages()返回的是加载类的类加载器定义之包(每个加载器都会有个未命名模块)。isExported()用来判断指定的包是否有 exports，它有另一个重载版本，用来判断指定的包是否 exports to 某个模块。

如果执行程序时指定命令行自变量为 java.lang.Object，那么会显示以下结果：

```
cc.openhome 模块可读取 java.lang.Object 类所在之 java.base 模块
exports 的包：
javax.net.ssl
```


严禁网络传播本PDF，违者责任自负！

```
java.util.stream
javax.crypto.interfaces
java.lang.invoke
javax.security.cert
java.util.regex
java.lang
java.lang.module
java.security.cert
java.nio.charset
java.security.spec
java.io
java.util.concurrent
java.time.temporal
...
```

除了通过 Module 检查模块信息之外，Module 还定义了 addExports()、addOpens()，可以增加模块 exports、opens 的包，而 addReads()可以增加模块 requires 的模块，addUses()可以增加模块使用到的服务类。

与 19.1.4 节介绍的--add-exports、--add-opens、--add-reads 等非标准自变量不同，只有在模块内的类，才能调用所在模块的 **addExports()、addOpens()、addReads()、addUses()**，因此，如果取得 java.base 模块的 Module 实例并调用其 addOpens()方法，将会引发 IllegalCallerException。

举个例子来说，在 cc.openhome.util 模块中，如果有个 Some 类如下：

ModuleAPI Some.java

```
package cc.openhome.util;

public class Some {
    private int some;

    public void openTo(Module module) {
        this.getClass().getModule().addOpens("cc.openhome.util", module);
    }
}
```

cc.openhome.util 模块的模块描述文件中，仅 exports 了 cc.openhome.util 包，其他模块基本上就只能存取公开的 API，然而 Some 类提供了 openTo()方法，可以将 cc.openhome.util 包 exports 给指定模块。如果 cc.openhome 模块中有下面的范例程序：

ModuleAPI OpensDemo.java

```
package cc.openhome;

import cc.openhome.util.Some;
import java.lang.reflect.Field;

public class OpensDemo {
    public static void main(String[] args) throws Exception {
        Some s = new Some();
        if(args.length != 0 && "opens".equals(args[0])) {
            s.openTo(OpensDemo.class.getModule());
        }
        Field f = Some.class.getDeclaredField("some");
```


严禁网络传播本PDF，违者责任自负！

```
        f.setAccessible(true);
        System.out.println(f.get(s));
    }
}
```

执行这个范例时，若没有指定命令行自变量，会因为试图存取私有值域而发生 InaccessibleObjectException，若执行时指定命令行自变量 opens，那么 cc.openhome.util 包会 opens 给 cc.openhome 模块，就能存取私有值域。

19.2.2 使用 ModuleDescriptor

Module 实例可以取得的模块信息其实有限，它提供了 **getDescriptor()** 可以返回 **java.lang.module.ModuleDescriptor**，这个实例代表着命名模块(包含自动模块)的模块描述文件，未命名模块的 **getDescriptor()** 会返回 **null**。

ModuleDescriptor 代表着模块描述文件，这意味着，它包含的是静态的模块描述信息，也就是模块描述文件的内容。举例来说，如果通过 Module 的 addOpens() 方法开放了更多包，那么 Module 的 getPackages() 方法返回的 Set<String>就会包含 addOpens() 方法开放的包，不过，ModuleDescriptor 的 packages() 方法返回的 Set<String>，仍旧是模块描述文件中定义有 exports 的包。

部署描述文件中的 requires、exports、opens 等语句，在 ModuleDescriptor 上有对应的 ModuleDescriptor.Requires、ModuleDescriptor.Exports、ModuleDescriptor.Opens 等类，而 ModuleDescriptor 的 requires()、exports()、opens() 等方法，返回的就是对应的 Set<ModuleDescriptor.Requires>、Set<ModuleDescriptor.Exports>、Set<ModuleDescriptor. Opens>等对象。

ModuleDescriptor 提供了静态的 read() 方法，可以读取并根据 module-info.class 的内容建立 ModuleDescriptor 实例。例如，读取 java.base 的 module-info.class：

```
Module m = Object.class.getModule();
ModuleDescriptor md = ModuleDescriptor.read(
    m.getResourceAsStream("module-info.class")
);
```

在这个片段中看到，如果要读取模块中的资源，可以通过 Module 的 getResourceAsStream() 方法。

ModuleDescriptor 提供了静态的 newAutomaticModule()、newModule()、newOpenModule() 等方法，可以来建立新的模块描述定义，它们都返回 ModuleDescriptor.Builder，因而可以流畅 API 风格来建立 ModuleDescriptor，在 ModuleDescriptor.Builder 的 API 文件中，提供了下面的范例：

```
ModuleDescriptor descriptor = ModuleDescriptor.newModule("stats.core")
        .requires("java.base")
        .exports("org.acme.stats.core.clustering")
        .exports("org.acme.stats.core.regression")
        .packages(Set.of("org.acme.stats.core.internal"))
        .build();
```

19.2.3 浅谈 ModuleLayer

Module 实例上定义了 **getLayer()** 方法，它会返回 **java.lang.ModuleLayer** 实例，代表该模块是在哪个模块层中找到的，那么什么是模块层？

严禁网络传播本PDF，违者责任自负！

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

回顾 17.2 节介绍过的类加载器，读者知道它们负责加载类，而在 JDK9 中，类必然属于某个模块，程序执行时，模块间的关系构成了模块图，类加载器必须在模块图中寻找类，模块图与类加载器的关系就组成了模块层。

到目前为止，范例程序运行时都只有一个模块层，称为 boot 模块层，而类加载器都是在 boot 模块层的模块图中寻找类，boot 模块层上还有个空模块层(Empty layer)，里面没有任何模块，主要目的是作为 boot 模块层的父模组层。

可以通过 **ModuleLayer.boot()** 来取得 boot 模块层的代表实例，由于目前类加载器都是在 boot 模块层中寻找类，因此通过 Module 的 getLayer() 取得的就是 boot 模块层：

```
jshell> ModuleLayer.boot() == Object.class.getModule().getLayer();
$1 ==> true
```

绝大多数的应用程序项目并不需要意识到模块层的存在，例如某种服务容器或者 plugins 架构的应用程序才会使用模块层。在 ModuleLayer 的 API 文件中提供了个简单的范例，大致介绍如何建构模块层，并从模块层中找到模块并加载类。

首先，使用 java.lang.module.ModuleFinder 来找寻模块，假设模块的 JAR 文件位于 One 项目 dist 文件夹：

```
ModuleFinder finder =
    ModuleFinder.of(Paths.get("C:\\workspace\\One\\dist"));
```

接下来解析模块建立模块图，解析模块是通过 java.lang.module.Configuration，由于 java.base 等模块位于 boot 模块层，在解析模块时会需要 boot 模块层的 Configuration，在取得 boot 模块层的 ModuleLayer 实例之后，可以调用 configuration() 来取得：

```
ModuleLayer boot = ModuleLayer.boot();
Configuration cf = boot.configuration()
    .resolve(finder, ModuleFinder.of(), Set.of("one"));
```

调用 resolve() 方法进行解析时，第一个参数指定从哪个 ModuleFinder 找寻模块，如果在父 ModuleFinder 与第一个参数指定的 ModuleFinder 都找不到模块的话，会使用第二个参数指定的 ModuleFinder，第三个参数指定模块层的根模块名称，One 项目 dist 文件夹中的 one.jar 中有个 one 模块，因此第三个参数使用了 Set.of("one")。

在解析完成取得 Configuration 实例之后，可以用该实例在 boot 模块层下建立新的模块层，建立时指定了 System 加载器作为父加载器：

```
ClassLoader scl = ClassLoader.getSystemClassLoader();
ModuleLayer layer = boot.defineModulesWithOneLoader(cf, scl);
```

如果要从新的模块层加载类，先找出模块的类加载器，使用该加载器进行加载：

```
Class<?> c = layer.findLoader("one")
                  .loadClass("cc.openhome.one.OneClass");
```

就这个例子来说，findLoader("one") 找到的加载器，父加载器会是 System 加载器，因此若在 System、Platform、Bootstrap 加载器架构下找不到类，才会由 findLoader("one") 找到的加载器来载入类。

以上通过 ModuleLayer 的 API 文件中提供的简单范例，稍微介绍了一下何谓模块层，更多细节，可以参考 API 文件中的介绍。

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

19.3 打包模块

在 1.2.4 节中讲过，想要打包模块，在 JDK9 中除了 JAR 之外，还有 JMOD 与 JIMAGE。建立这些模块打包，应该要使用更为便捷的工具程序来代劳，不过通过几个简单的范例，稍微介绍一下如何使用 JDK 内建指令来建立 JAR、JMOD 与 JIMAGE，也能比较清楚那些打包模块时更为便捷的工具程序到底做了哪些事情。

19.3.1 使用 jar 打包

身为 Java 开发者，JAR 是再熟悉也不过的格式了，它采用 zip 压缩格式，可以通过 JDK 内建的 `jar` 指令建立，如果使用 IDE 或者是 Maven、Gradle 之类的构建工具(Build tool)，都有更便捷的指令代劳 JAR 文件的建立。

1. 基本 JAR 打包

如果想体验一下如何使用 `jar` 指令自行建立 JAR 文件，读者可以下载范例文件中 labs\CH19\Hello2 文件夹复制至 C:\workspace。Hello2 是 2.3.1 节的范例成果，其中 classes 文件夹含有 `cc.openhome.Main` 类的编译结果，mods\cc.openome 则是 `cc.openhome` 模块的编译结果。

首先，试着建立一个传统的类库 JAR 封装，在进入 Hello2 文件夹之后，执行如下指令：

```
C:\workspace\Hello2>jar --create --file dist/helloworld.jar -C classes /

C:\workspace\Hello2>java -cp dist/helloworld.jar cc.openhome.Main
Hello, World
```

`--create` 表示建立 JAR，`--file` 指定了 JAR 文件名，`-C` 自变量用来指定 JAR 要包含哪个文件夹的内容，除了使用`-cp` 来指定程序进入点类之外，如果在建立 JAR 文件时，使用 `--main-class` 指定了程序进入点类，那么可以使用`-jar` 来直接执行 JAR 文件。例如：

```
C:\workspace\Hello2>jar --create --file dist/helloworld2.jar --main-class cc.openhome.
Main -C classes /

C:\workspace\Hello2>java -jar dist/helloworld2.jar
Hello, World
```

Hello2 的 mods 中，cc.openhome 文件夹有编译好的 `cc.openhome` 模块，可以使用下面的指令来建立模块 JAR：

```
C:\workspace\Hello2>jar --create --file dist/cc.openhome.jar --main-class cc.openhome.
Main -C mods/cc.openhome /

C:\workspace\Hello2>java --module-path dist/cc.openhome.jar -m cc.openhome/cc. openhome.
Main
Hello, World

C:\workspace\Hello2>java --module-path dist/cc.openhome.jar -m cc.openhome
Hello, World
```

这次在建立 JAR 时，同时指定了`--main-class`，因此可以在使用`-m` 时只指定模块名称，就

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

显示了“Hello, World”。

如果想要查看 JAR 中有哪些文件，可以使用--list 自变量，例如：

```
C:\workspace\Hello2>jar --list --file dist/cc.openhome.jar
META-INF/
META-INF/MANIFEST.MF
module-info.class
cc/
cc/openhome/
cc/openhome/Main.class
```

2. 多版本 JAR

JDK9 增强了 JAR 格式，可以建立多版本 JAR(Multi-release JAR)，其中可以包含为多个 JDK 版本下编译的.class 文件。如果运行在 JDK8 或旧版本时，就使用旧版本的.class；如果是运行在 JDK9 或之后版本 JDK，就使用新版本的.class。

请将范例文件中 labs\CH19\MultiReleases 文件夹复制至 C:\workspace，MultiReleases 中包含了 classes 文件夹，其中的 Main.class 是在 JDK8 下编译 src\cc\openhome\Main.java 而来，如果用它作为程序进入点执行，会显示“Hello, JDK8”；MultiReleases 中包含了 mods 文件夹，其中的 Main.class 是在 JDK9 中编译 src\cc.openhome\cc\openhome\Main.java 而来，如果用它作为程序进入点执行，会显示“Hello, World”。

现在可以进入 MultiReleases 文件夹，并执行如下指令：

```
C:\workspace\MultiReleases>jar --create --file dist/cc.openhome.jar --main-class
cc.openhome.Main -C classes / --release 9 -C mods/cc.openhome /
```

留意到--release 之前的内容，这与旧式的 JAR 建立方式无异，因此，指定要包装的内容会是 JDK8 或更旧版下编译好的.class 文件来源，--release 指定了 9，紧接着的-C 指定的是 JDK9 中编译好的.class 文件来源。现在，来看看 JAR 中有什么了：

```
C:\workspace\MultiReleases>jar --list --file dist/cc.openhome.jar
META-INF/
META-INF/MANIFEST.MF
META-INF/versions/9/module-info.class
cc/
cc/openhome/
cc/openhome/Main.class
META-INF/versions/9/
META-INF/versions/9/cc/
META-INF/versions/9/cc/openhome/
META-INF/versions/9/cc/openhome/Main.class
```

若不看粗体字的部分，实际上布局与旧式 JAR 无异，这部分的信息主要来自使用 jar 指令时--release 之前的指定，因此在 JDK8 上使用这个 JAR 时，如果是 java -jar dist/cc.openhome.jar 的话，会显示“Hello, JDK8”。

至于粗体字的部分，会在 JDK9 上使用该 JAR 文件时才会有作用。例如在 JDK9 执行 java -jar dist/cc.openhome.jar 的话，若发现 JAR 根目录的.class 与 META-INF/versions/9 中的.class 没有重复，会使用 JAR 根目录中的.class；如果 META-INF/versions/9 中有重复的.class，或者 JAR 根目录中不存在 META-INF/versions/9 中的.class(例如 module-info.class)，就使用 META-INF/versions/9 中

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

的.class。

因此，在 JDK9 中执行如下指令的话，会显示“Hello, World”。

```
C:\workspace\MultiReleases>java -jar dist/cc.openhome.jar
Hello, World

C:\workspace\MultiReleases>java --module-path dist/cc.openhome.jar -m cc.openhome
Hello, World
```

由于 META-INF/versions/9 的 cc/openhome/Main.class 与 JAR 根目录的 cc/openhome/Main.class 重复，在 JDK9 上会使用 META-INF/versions/9 的 cc/openhome/Main.class。由于 META-INF/versions/9/module-info.class 不存在于 JAR 根目录，在 JDK9 上通过--module-path 与-m 也是可以执行的。

提示>>> 这里对 jar 指令做了基本的介绍，若读者想了解更多，可以参考 jar 的文件，链接地址如下：docs.oracle.com/javase/9/tools/jar.htm

19.3.2 使用 jmod 打包

JDK9 引进了新的 JMOD 来打包模块，目的在于可以处理比 JAR 更多的文件类型，例如原生指令、类型文件等。有些文件将 JMOD 描述为扩充版本的 JAR，使用 javac 或 jlink 工具程序时，可以指定--module-path 自变量至.jmod 文件所在路径，然而执行时期不支持 JMOD 文件。

可以使用 JDK9 内置的 **jmod** 工具程序来建立、查看或取出 JMOD 文件的内容，在撰写本章的这个时间点，jmod 建立的文件采用 zip 压缩格式，然而这只是暂时性的方案，未来的格式仍是个开放讨论的议题。

由于执行时期不能使用 JMOD，发布给开发者使用的链接库时，仍是以 JAR 为主，jmod 工具程序也能够使用模块 JAR 文件作为.class 来源建立 JMOD，发布 JMOD 文件的时机在于，需要一并打包原生指令、类型文件等，让客户端能便于自定义执行时期映像文件；如 1.2.4 节介绍过的，JDK9 本身提供了 Java SE API 的 JMOD 文件，可以在 JDK9 安装文件夹的 jmods 文件夹中找到。

1. 基本 JMOD 打包

Lab

如果要体验一下如何使用 jar 指令自行建立 JAR 文件，可以在 19.3.1 节使用过的 Hello2 文件夹进行操作。例如，要将 mods/cc.openhome 中的 cc.openhome 模块，封装为 cc.openhome.jmod 的话，可以输入如下指令：

```
C:\workspace\Hello2>jmod create --class-path mods/cc.openhome --main-class cc.openhome.
Main dist/cc.openhome.jmod

C:\workspace\Hello2>jmod list dist/cc.openhome.jmod
classes/module-info.class
classes/cc/openhome/Main.class
```

--class-path 用来指定.class 来源，来源也可以是个 JAR 文件，--main-class 指定程序进入点类(如果有的话)，这会在 dist 文件夹中建立 cc.openhome.jmod。想要查看 JMOD 的内容，可以使用 list 自变量，Java 的.class 文件会被放置在 classes 之中。如果打包时有可执行文件(使用--cmds)会放在 bin，类型文件(使用--config)放在 conf，头文件(使用--header-files)放在

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

include，原生链接库(使用--lib)放在 lib，版权信息(使用--legal-notices)放在 legal。例如，查看一下 JDK9 中 jmods 文件夹的 java.base.jmod：

```
C:\workspace\Hello2>jmod list "C:\Program Files\Java\jdk-9\jmods\java.base.jmod"
classes/module-info.class
classes/com/sun/crypto/provider/AESCipher$AES128_CBC_NoPadding.class
classes/com/sun/crypto/provider/AESCipher$AES128_CFB_NoPadding.class
...
conf/net.properties
conf/security/java.policy
conf/security/java.security
...
include/classfile_constants.h
include/jni.h
include/jvmti.h
...
legal/COPYRIGHT
legal/LICENSE
legal/public_suffix.md
...
bin/java.exe
bin/javaw.exe
bin/keytool.exe
lib/classlist
lib/java.dll
lib/jimage.dll
lib/jli.dll
...
```

稍后介绍 jlink 时，会使用刚建立好的 cc.openhome.jmod 文件，接下来先看看如何记录 JMOD 文件的哈希值。

2. 记录 JMOD 哈希值

如果其他模块依赖在目前正在打包的模块，可以记录可依赖在此打包模块的其他模块哈希值，一旦记录了哈希值，在使用 **jlink** 时会验证哈希值是否符合，若不符合建立就会失败并显示错误信息。

为了实际操作，可以使用 2.3.3 节的范例成果，请复制范例文件中 labs\CH19\Hello3 文件夹至 C:\workspace，在进入 Hello3 文件夹之后，使用下面的指令为 cc.openhome 及 cc.openhome.util 模块分别建立 cc.openhome.jmod 与 cc.openhome.util.jmod。

```
C:\workspace\Hello3>jmod create --class-path mods/cc.openhome --main-class cc.openhome.
Main dist/cc.openhome.jmod

C:\workspace\Hello3>jmod create --class-path mods/cc.openhome.util dist/cc.openhome.
util.jmod
```

接着，执行 jmod 指令时指定 describe，先查看一下两个.jmod 的信息。

```
C:\workspace\Hello3>jmod describe dist/cc.openhome.jmod
cc.openhome
requires cc.openhome.util
requires java.base mandated
contains cc.openhome
```


非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
main-class cc.openhome.Main

C:\workspace\Hello3>jmod describe dist/cc.openhome.util.jmod
cc.openhome.util
exports cc.openhome.util
requires java.base mandated
```

目前没有记录任何的哈希值，接下来使用 jmod hash 为模块加入哈希信息。

```
C:\workspace\Hello3>jmod hash --module-path dist --hash-modules .*
Hashes are recorded in module cc.openhome.util

C:\workspace\Hello3>jmod describe dist/cc.openhome.jmod
cc.openhome
requires cc.openhome.util
requires java.base mandated
contains cc.openhome
main-class cc.openhome.Main

C:\workspace\Hello3>jmod describe dist/cc.openhome.util.jmod
cc.openhome.util
exports cc.openhome.util
requires java.base mandated
hashes cc.openhome SHA-256a7ae8af153d2210c54f455b8cdb19ae2155f4d895ab256e502ed9cb5
ea2ee51a
```

--hash-modules 用来指定为哪些模块间的依赖关系记录哈希值，使用的是规则表示式，可以看到在 cc.openhome.util.jmod 上记录了 cc.openhome 的哈希信息。

为了稍后在使用 jlink 时作为对照，也执行如下指令，在 dist2 中另外建立一个 cc.openhome.jmod：

```
C:\workspace\Hello3>jmod create --class-path mods/cc.openhome --main-class cc.openhome.
Main dist2/cc.openhome.jmod
```

提示>>> 这里对 jmod 指令做了基本的介绍，若读者想了解更多，可以参考 jmod 的文件，链接地址如下：docs.oracle.com/javase/9/tools/jmod.htm

19.3.3 使用 jlink 建立执行时期映像

可以使用 JDK9 附带的 jlink 工具程序，建立一个执行时期映像，其中只包含指定的模块，这意味着执行特定的 Java 应用程序时，不需要完整的标准 JRE(JRE9 是二百多 MB)，若基于标准 JDK，基本“Hello, World”程序的执行时期映像会是三十几 MB，通过压缩、去除不必要的头文件等设定，容量还可以更小。

来看看实际上如何使用 jlink。首先使用 19.3.2 节的 Hello2 成果，从 cc.openhome.jmod 与 java.base.jmod 中建立执行时期映像，在进入 Hello2 文件夹后，执行如下指令：

```
C:\workspace\Hello2>jlink --module-path "C:\Program Files\Java\jdk-9\jmods;dist\cc.
openhome.jmod" --add-modules cc.openhome --output helloworld
```


严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！


```
C:\workspace\Hello2>helloworld\bin\java --list-modules
cc.openhome
java.base@9

C:\workspace\Hello2>helloworld\bin\java -m cc.openhome
Hello, World
```

--module-path 指定了模块来源，来源可以是目录、JAR 或 JMOD，--add-modules 指定要将哪些模块加入为根模块，--output 指定输出至哪个文件夹。只有使用到的模块才会被加入执行时期映像，就 Hello2 来说，只会使用到 java.base 与 cc.openhome 模块。

由于 java.base.jmod 包含 java 等指令，在执行时期映像建立好之后，输出文件夹中的 bin 里就会有 java 等指令，使用 java --list-modules 可以列出执行时期映像中的模块。由于先前 cc.openhome.jmod 在建立时，使用--main-class 指定了程序进入点类，因此可以直接使用 java -m cc.openhome 来执行程序。

在 19.3.2 节最后曾为 JMOD 记录了哈希值，目前在 Hello3 文件夹中，dist/cc.openhome.util.jmod 必须搭配 dist/cc.openhome.jmod，在使用 jlink 建立执行时期映像时，才能通过哈希值的比对。例如，进入 Hello3 数据之后，执行如下指令：

```
C:\workspace\Hello3>jlink --module-path "C:\Program Files\Java\jdk-9\jmods;dist"
--add-modules cc.openhome --output helloworld

C:\workspace\Hello3>helloworld\bin\java --list-modules
cc.openhome
cc.openhome.util
java.base@9

C:\workspace\Hello3>helloworld\bin\java -m cc.openhome
Hello, World
```

如果故意在建立执行时期映像时，使用 dist2/cc.openhome.jmod 的话，就会因为哈希值比对时不符合而失败。

```
C:\workspace\Hello3>jlink --module-path "C:\Program Files\Java\jdk-9\jmods;dist2\cc.
openhome.jmod;dist\cc.openhome.util.jmod" --add-modules cc.openhome --output helloworld
Error: Hash of cc.openhome (9242a4355bef706c421e6fe9e7b3db6805c2f11fb58f6ca23f90d35
1018abc69) differs to expected hash (a7ae8af153d2210c54f455b8cdb19ae2155f4d895ab256
e502ed9cb5ea2ee51a) recorded in cc.openhome.util
```

提示 »» 这里对 jlink 指令做了基本的介绍，若读者想了解更多，可以参考 jlink 的文件，链接地址如下：docs.oracle.com/javase/9/tools/jlink.htm

19.4 重点复习

Java SE 9 也将 JDK 本身做了模块化，src.zip 中的原始码被划分到各个模块，除了可使用 JAR 来封装模块之外，JDK9 还有 JIMAGE 与 JMOD 格式。

在类路径下被发现的类，都会被自动归类到未命名模块，在模块路径下被发现的类，则都属于某个命名模块。如果应用程序采取模块化设计，默认就会依赖在 java.base 模块之上，

严禁网络传播本PDF，违者责任自负！ 非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

java.base 模块 exports 了它全部的包，任何模块都可以读取 java.base 模块。

在 JDK9 之后多了模块，模块中的 public 类、方法、值域是否能被另一模块看见，还要视模块设定而定。

在定义模块时必须记得，同一个包不可以同时出现在被 requires 的两个模块中，这称为分裂包会导致编译失败，而执行时期会产生 LayerInstantiationException。

就算是 Java SE API，若不是 java.base 模块中的类，还是必须 requires 相关的模块。

明确定义了模块并放在模块路径中，这样的模块称为显式模块，显式模块无法依赖在未命名模块，因为未命名模块并没有名称，无法在显式模块的模块描述文件中进行 requires。

若是采用模块化设计，可以将未支持模块化设计的JAR文件放在模块路径，这会使得该JAR文件被视为自动模块，自动模块也是命名模块的一种。模块有其名称产生的规则，有了模块名称之后，就可以 requires 自动模块，也就可以使用自动模块中公开的类、方法与值域。

基于兼容性，自动模块有隐含的模块定义，可以读取其他模块，其他模块也可以读取自动模块。应用程序在迁移至模块化设计的过程中，自动模块会是未命名模块至显式模块之间的桥梁。

并不是任何 JAR 都可以自动产生正确的模块名称，默认是基于文件名，若被放到模块路径的 JAR 文件，无法自动产生正确的模块名称，编译时期没有名称可以 requires，而执行时期会产生 IllegalArgumentException，从而使得 JVM 无法初始模块层而发生 FindException。

如果同一个包出现在模块路径上多个已有的 JAR 文件中，由于一个包不能出现在多个模块，因此只有其中一个 JAR 能成为自动模块，而其他会被忽略。

如果要能操作非公开的类、方法或值域，模块本身得 opens 指定的包，或者直接 open 模块本身。

java.base 模块 exports 了所有的包，但没有 opens 包，也没有 open 模块本身，因此，任何模块都不能对 java.base 做深层反射。

未命名模块可以读取其他模块，然而能不能存取或深层反射，要视其他模块是否 exports 或 opens。而未命名模块对其他模块来说，则如 open 全部包，因此执行时期，任何模块都可以存取与深层反射未命名模块。不过，因为未命名模块没有名称，因此显式模块无法 requires，也就不能存取(也不能读取)未命名模块。

自动模块可以读取其他模块，然而能不能存取或深层反射，要视其他模块是否 exports 或 opens。自动模块对其他模块来说，如同 open 全部包，因此可以存取与深层反射自动模块，自动模块是命名模块，显式模块也可以 requires 自动模块。

被弃用的模块可以使用@Deprecated 标注，某个模块如果还是必须使用被弃用的模块，又不想收到警告，那就加上个@SuppressWarnings("deprecation")。

19.5 课后练习

在范例文件的 samples\CH19 中有个 JNotePad 项目，这是旧版 JDK 开发的简单记事本程序，请将项目模块化，并建立执行时期映像(输出至 JNotePad 项目中的 JNotePad 文件夹)，并可以如下执行程序：

```
C:\workspace\JNotePad>JNotePad\bin\javaw -m JNotePad
```


严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

Chapter A 如何使用本书项目

学习目标

- 范例项目环境配置
- 范例项目导入

严禁网络传播本PDF，违者责任自负！

非卖品，仅供非商业用途或交流学习使用

版权所有，严禁以任何方式传播本PDF，违者自负法律责任！

A.1 项目环境配置

为了方便读者查看范例程序、运行范例以观摩成果，书附范例文件提供了本书每个章节范例。由于每个读者的计算机环境配置不尽相同，在这里对本书范例制作时的环境加以介绍，以便读者能够配置出与作者制作范例时最为接近的环境。

本书撰写过程安装的软件：

- Oracle JDK9
- NetBeans IDE 最新开发版本（Development Build）

JDK9 的下载、安装，请见 1.2.3 节，NetBeans IDE 的安装，请见 2.4.1 节。

跟安装及路径有关的信息包括：

- JDK 安装在 C:\Program Files\Java\jdk-9.0.4 文件夹，PATH 环境变量中包括 C:\Program Files\Java\jdk-9.0.4\bin 文件夹。
- NetBeans IDE 案例都是建在 C:\workspace 文件夹中。

A.2 打开案例

如果要使用范例文件的范例项目，首先将想使用的范例项目复制至 C:\workspace 中，接着在 NetBeans 中执行打开项目的操作：

(1) 选择“文件”|“打开项目”命令，打开“打开项目”对话框，选择项目文件夹（会有个或者是图标，前者是 JDK9 模块项目或者是基于类别路径的项目）。

(2) 单击“打开项目”按钮，即可打开项目。

如果打开项目后，发现如图 A.1 所示对话框，可能是链接库相对路径不符，必须调整相关链接库。

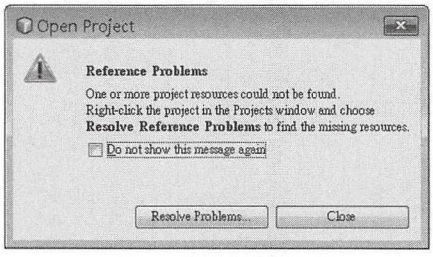


图 A.1 发生相依性问题

(1) 单击图 A.1 中 Resolve Problems 按钮。

(2) 通过 Reference Problems 下的内容了解有相关问题的链接库，单击 Resolve 按钮进行链接库的调整(寻找 JAR 文档或放置.class 的文件夹)。

其他特定范例的设定，请参考各章节中的操作步骤说明。

严禁网络传播本PDF，违者责任自负！

- 本书是作者多年来教学实践经验的总结，汇集了学员在学习JSP & Servlet或认证考试时遇到的概念、操作、应用等各种问题及解决方案
- 基于Servlet 4.0/Java SE 8重新改版，无论章节架构还是范例程序代码，都做了全面更新
- 详细介绍了OWASP TOP10、CWE、CVE，讨论了会话安全、密码管理、Java EE安全机制、CSRF等Web安全基本概念
- 增加了对Spring、Spring MVC、Spring Boot的入门介绍，认识Web MVC框架与快速开发工具的使用
- 涵盖了文本处理、图片验证、自动登录、验证过滤器、压缩处理、线上文件管理、邮件发送等实用范例
- 以“微博”项目贯穿全书，将JSP & Servlet技术应用于实际项目开发之中，并使用重构方式来改进应用程序架构
- 提供练习的Lab操作文档，方便读者掌握练习重点，将IDE操作纳入教学内容，使读者能与实践结合
- 本书适合JSP & Servlet初学者以及广大JSP & Servlet技术应用人员

JDK 9

Java 学习笔记

- 本书是作者多年来教学实践经验的总结，汇集了学员在学习课程或认证考试中遇到的概念、操作、应用等问题及解决方案
- 针对Java SE 9新功能全面改版，无论是章节架构或范例程序代码，都做了重新编写与全面翻新
- 详细介绍了Java 9模块化后，JVM、JRE、Java SE API、JDK与IDE之间的对照关系
- 从Java SE API的源代码分析，了解各种语法在Java SE API中的具体应用
- 提供练习的Lab操作文档，方便读者掌握练习重点
- 将IDE操作纳入教学内容，使读者能与实践结合

资源下载：

- 各章NetBeans范例项目
- 各章Lab的NetBeans项目
- 教学PPT

清华社官方微信号

扫 我 有 惊 喜

ISBN 978-7-302-50118-3

9 787302 501183 >

定价：98.00元